**Salesforce Interview Question**

**PLC:**[**https://partnerlearningcamp.salesforce.com/s/learner-dashboard**](https://partnerlearningcamp.salesforce.com/s/learner-dashboard)

**In Salesforce, this is the order in which actions execute:**1.Process Start  
2. System Validation Rules

3Record Triggered flow  
4. Apex Before Triggers  
5. Custom Validation Rules  
6. Duplicate Rules

7.Save the Record [No Commit]  
8. Apex After Triggers  
9. Assignment Rules  
10. Auto-Response Rules  
11. Workflow Rules  
12. Processes and Flow  
13. Escalation and Entitlement  Rules

14.Record Triggered Flow   
15. Roll-Up Summary Fields

16.Sharing Rule evaluation

17.Commit all DML Changes

18.Post Commit Operation

**2. How is LWC different from Aura?**

LWC-based lightning components are built using web stack tools, whereas aura-based lightning components are built using [HTML5](https://www.simplilearn.com/tutorials/html-tutorial/html-vs-html5) and JavaScript tools.

Aura components **can** contain Lightning web components.

Lightning web components can’t contain Aura components.

when we communicate with Parent to child then all attribute convert into **cababe case**( simpleMessage --> simple-message)

**Why does LWC work faster than Aura components?**

Normally, the lightning web components work faster than aura components. This is because due to the absence of an abstraction layer

**LWC lifecycle hook methods**

1. constructor()
2. connectedCallback()
3. renderedCallback()
4. render()
5. disconnectedCallback()
6. errorCallback(error, stack)

**How to use Date Field in APEX Trigger**

* Opp.CloseDate= System.today();  
  Opp.CloseDate= System.today()+ 5; ==> date+5days  
  Opp.CloseDate = Date.newInstance(2017, 12, 31);  
  Opp.CloseDate = Date.parse('10/14/2017');  
  Opp.CloseDate = Date.valueOf('2017-12-15');

**Setup Context**

* **Definition**: The setup context refers to the administrative and configuration side of Salesforce. This includes tasks like creating objects, fields, workflows, validation rules, and other configurations.
* **Access**: Only users with administrative privileges can access the setup area.
* **Examples**:
  + Creating custom objects and fields.
  + Setting up user roles and profiles.
  + Configuring security settings and permissions.
  + Managing data import and export tools.

**Non-Setup Context**

* **Definition**: The non-setup context refers to the regular user interface where users interact with Salesforce data and perform their daily tasks.
* **Access**: All users, depending on their permissions, can access the non-setup area.
* **Examples**:
  + Creating and managing records (e.g., Accounts, Contacts, Opportunities).
  + Running reports and dashboards.
  + Using Salesforce features like Chatter, Tasks, and Events.
  + Interacting with custom applications and components.

**With and Without Sharing:** <https://www.youtube.com/watch?v=ZJjXxAl260s>

By default Without Sharing is implemented.

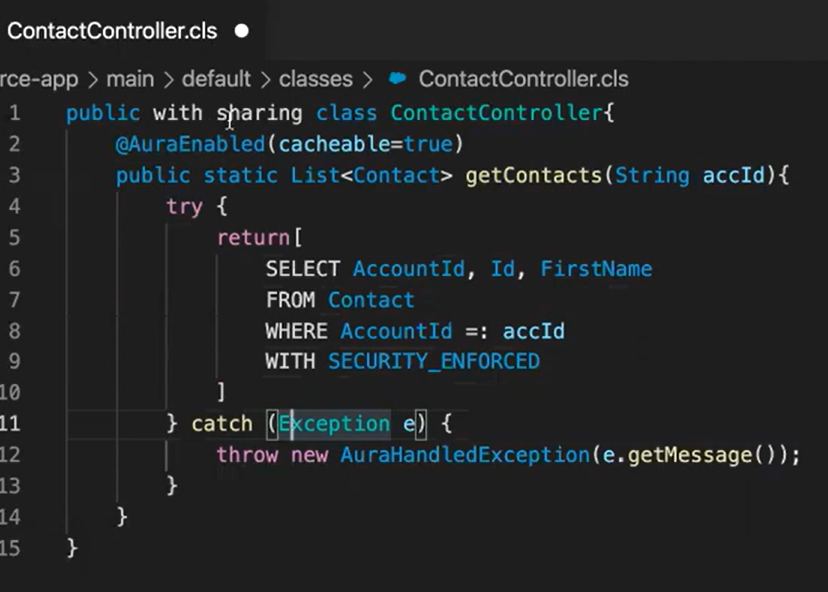
**Without Sharing**

* **Ignores sharing rules**: Declaring a class with the without sharing keyword means that the class does not enforce the sharing rules of the current user. This allows the class to access all records, regardless of the user’s permissions.

**With Sharing**

* **Enforces sharing rules**: When a class is declared with the with sharing keyword, it enforces the sharing rules of the current user. This means that the class respects the user’s permissions and only allows access to records that the user has permission to view or modify.

**With Sharing:** works for Record Label Sharing if we want to apply Object and Fields label Security we have to add one condition **WITH SECURITY\_ENFORCED**



**custom setting(\_\_c) and custom metadata(\_\_mdt)**  
**custom Setting:** are two types list and hierarchy store in cache memory but not able to deployed across environments.  
1.No query required  
2.fast Access of data increase the performance  
3. Hierarchical custom setting allow different data for different profile/user.

1.custom setting data cannot be deployed  
2. only 300 files allow custom setting  
3. custom setting counts against total no of custom object in organization.  
4. Tabs cannot be created in custom setting.

**custom metadata:** are like list and we are able to deploy it from one org to another org by chaneSet. and it can be deployed as a packaged.  
1.Validation Rules  
2.page Layouts  
3. we can create lookup to another custom metadata like custom object

1. like custom setting custom metadata also doesn't have an option to create custom tab.  
2. Need to Query to get data

| **Class** | **Interface** |
| --- | --- |
| The keyword used to create a class is “class” | The keyword used to create an interface is “interface” |
| A class can be instantiated i.e., objects of a class can be created. | An Interface cannot be instantiated i.e. objects cannot be created. |
| It can be inherited by another class using the keyword ‘extends’. | It can be inherited by a class by using the keyword ‘implements’ and it can be inherited by an interface using the keyword ‘extends’. |

**How to use Class inside omnistudio**

In **VlocityOpenInterface,** the invokeMethod returns a Boolean.

In **VlocityOpenInterface2**, the invokeMethod returns an object.

@JsonAccess(serializable='always')

**global with sharing class TestReserializeApex implements vlocity\_ins.VlocityOpenInterface2 {**

**global Object invokeMethod(String methodName, Map input, Map output, Map options) {**

return output.put('result', new TestReserialize());

**}**

@JsonAccess(serializable='always')

global public class TestReserialize {

          global public String name = 'Dave Smith';

           }

}

**Or**

global class **OmniStudioTest** implements **omniStudio.VlocityOpenInterface2**{    public Integer **invokeMethod(String methodName, Map<String,Object> input, Map<String,Object> output, Map<String,Object> options){**        Integer a=10;  
      return a;  
    }

}

**inside meta.xml**

**<runtimeNamespace>omnistudio</runtimeNamespace>**

**inside js**

**import { OmniscriptBaseMixin } from "omnistudio/omniscriptBaseMixin";**

**export default class ShowField  extends OmniscriptBaseMixin(LightningElement)**

**import getAccounts from '@salesforce/apex/AccountService.getAccounts';**

**How to create table by the help of Omniscript:** [**Edit Block in OmniScript**](https://www.omnistudiofacts.com/post/edit-block-in-omniscript)

Default Class for create, edit and delete records in salesforce: DefaultOmniScriptEditBlock

**To find login used Id: UserInfo.getUserId();**

**LWC Code**

import { LightningElement, track,wire, api } from 'lwc';

**import { getRecord } from 'lightning/uiRecordApi';**

const FIELDS= ['VoiceCall.VendorCallKey'] ; *// getting vendercall key value*

export default class AgentHelp extends LightningElement {

   @api recordId;

   @track vendorCallKeyFromVCR; *// VendorCallKey from Voice Call Record*

   @wire(getRecord, { recordId:'$recordId', fields:FIELDS})

   voiceCall({data,error}){

       console.log('Record ID:', this.recordId);

       if(data){

           console.log('BG data ', data);

           this.vendorCallKeyFromVCR = data.fields.VendorCallKey.value;

       }

       else{

           console.error(error)

       }

   }

}

* **Workflow Rule:** A workflow rule in Omnistudio is a declarative tool used to automate standard internal procedures and processes to save time across an organization. It primarily performs actions based on certain criteria and can create tasks, send email alerts, or update fields.
* **Process Builder:** The Process Builder is a more advanced tool that allows for the automation of complex business processes. It’s used to create processes with multiple criteria, multiple actions, and the ability to initiate flows and Apex code. It provides greater flexibility and control compared to workflow rules.

**While both tools are used for automation, the Process Builder is typically preferred for complex scenarios that involve multiple criteria and actions, whereas workflow rules are more suitable for simpler automation tasks.**

**Sharing Settings**

**Sharing settings define the baseline level of access to records for your entire organization. These settings are configured at the object level and include:**

* **Organization-Wide Defaults (OWD):** These settings determine the default access level for records that users do not own. For example, you can set the OWD for an object to Private, Public Read Only, or Public Read/Write.
* **Role Hierarchy:** This structure automatically grants access to records owned by users lower in the hierarchy to their superiors. It ensures that managers can access the records of their subordinates.
* **Manual Sharing:** Record owners can manually share records with other users who need access but do not have it through other sharing mechanisms.

**Sharing Rules**

**Sharing rules are used to extend access to records beyond what is specified by the organization-wide defaults. They allow you to make exceptions and grant additional access based on specific criteria:**

* **Owner-Based Sharing Rules:** These rules grant access to records based on the record owner. For example, you can create a rule that shares records owned by users in a specific role with users in another role.
* **Criteria-Based Sharing Rules:** These rules grant access based on field values in the records. For example, you can create a rule that shares all records where the “Status” field is “Approved” with a specific group of users.

**Example**

* **Organization-Wide Default: Set the OWD for the Account object to Private.**
* **Role Hierarchy: Ensure that managers can access their team’s accounts.**
* **Sharing Rule: Create a criteria-based sharing rule to share all accounts with the “Industry” field set to “Technology” with the Sales team.**

**Queues: Focus on workload management and record ownership.**

**Groups: Focus on record sharing and collaboration.**

**Q16. How does Omnistudio handle integration with external systems or data sources?  
Ans: Omnistudio provides several mechanisms for integrating with external systems and data sources:**

* **APIs:** Omnistudio offers various APIs, including REST and SOAP, to facilitate communication with external systems. Custom APIs can also be built to expose specific functionality.
* **Middleware:** Integration middleware, such as Salesforce Connect and MuleSoft, can be used to connect Omnistudio with a wide range of external systems and databases.
* **External Objects**: Salesforce Connect allows you to create external objects that represent data in external systems as if they were standard Salesforce objects, enabling seamless data access.
* **Data Import/Export:** Tools like Data Loader and Data Import Wizard support data exchange between Omnistudio and external data sources.
* **Platform Events:** Omnistudio can publish and subscribe to platform events to enable event-driven integration with external systems.
* **Heroku Integration:** For more advanced integration scenarios, Salesforce Heroku can be used to build custom applications that integrate with Omnistudio.

**These integration options ensure that Omnistudio can easily connect with external systems, enabling a unified view of data and processes.**

**Q22. What is the importance of testing and debugging in Omnistudio development, and what tools are available for these tasks?  
Ans: Testing and debugging are crucial aspects of Omnistudio development to ensure the quality and reliability of applications. Here’s their importance and the available tools:**

* **Importance of Testing:** Testing helps identify and rectify issues, bugs, and errors in the application code and configurations before they reach production, ensuring a smoother user experience and preventing data corruption**.**
* **Importance of Debugging:** Debugging is the process of identifying and fixing issues in the code or configurations. It helps developers pinpoint the root cause of problems and correct them efficiently.

**Available Tools:**

* **Salesforce Inspector:** A Chrome extension that provides a suite of tools for debugging Salesforce applications, including viewing component attributes and inspecting network requests.
* **Apex Debugger:** Integrated into Salesforce, it allows developers to set breakpoints and step through Apex code to analyze and fix issues.
* **Developer Console:** Salesforce’s built-in tool for debugging and monitoring application behavior, including logs, queries, and error messages.
* **Lightning Component Inspector:** Helps inspect and debug Lightning components within Lightning Experience.
* **Testing Frameworks:** Salesforce offers various testing frameworks like Apex testing and Lightning Component testing for automated testing of code and components.
* **Static Code Analysis Tools**: Tools like Salesforce PMD and Checkmarx can analyze code for security vulnerabilities and compliance issues.
* **System Log**: The system log in Salesforce provides detailed information about code execution, including governor limits, SOQL queries, and DML operations.

**Effective testing and debugging practices, along with the use of these tools, are essential for delivering robust and reliable Omnistudio applications.**

**Use the Action Debugger: The Action Debugger in OmniStudio is a powerful tool. It lets you check each service call, ensuring the correct data is sent and the expected results are returned.**

**Check Data JSON: Use the Data JSON tab to view the data structure formed by the JSON code. This helps you confirm the keys and their values.**

**Apex Debug Logs: If your OmniScript includes Apex elements, you can create an Apex class to perform debug activities.**

**link for Apex debug log:**[**Unable to see the data JSON of a running OmniScript (salesforce.com)**](https://help.salesforce.com/s/articleView?id=000391069&type=1)

**Omniscript is easy to use and Build in compare of Angular.**

**Key Differences Between Angular and LWC OmniScript**

1. **Framework:**
   * Angular: A full-fledged framework with a wide range of features and tools.
   * LWC: A lightweight framework specifically designed for Salesforce, focusing on performance and ease of use within the Salesforce ecosystem.
2. **Component Model:**
   * Angular: Uses a component-based architecture with modules, components, and services.
   * LWC: Also uses a component-based architecture but is more tightly integrated with Salesforce’s data and event handling.
3. **Data Binding:**
   * Angular: Supports two-way data binding.
   * LWC: Primarily uses one-way data binding, with two-way data binding available through specific mechanisms.
4. **Performance:**
   * Angular: Can be heavier due to its extensive features.
   * LWC: Optimized for performance within the Salesforce platform, with a smaller footprint.
5. **Integration:**
   * Angular: Requires more effort to integrate with Salesforce.
   * LWC: Seamlessly integrates with Salesforce, leveraging native Salesforce features and APIs.

**reference:** [**Convert an Angular OmniScript to an LWC OmniScript (salesforce.com)**](https://help.salesforce.com/s/articleView?id=sf.os_convert_an_angular_omniscript_to_an_lwc_omniscript.htm&type=5)

**How to know about Recursive Trigger**

1. CANNOT\_INSERT\_UPDATE\_ACTIVATE\_ENTITY: This error often indicates that an update operation is causing another trigger to fire, leading to a potential infinite loop.
2. Duplicate id in list: This suggests that the same Account ID is being added multiple times to the list of accounts to update, which can happen if the trigger runs recursively.

**Error Message**

**conDesc: execution of AfterUpdate caused by: System.DmlException: Update failed. First exception on row 0 with id 0015g00001P75lDAAR; first error: CANNOT\_INSERT\_UPDATE\_ACTIVATE\_ENTITY, DescCon: execution of AfterUpdate caused by: System.DmlException: Update failed. First exception on row 0 with id 0035g000013rbvDAAQ; first error: CANNOT\_INSERT\_UPDATE\_ACTIVATE\_ENTITY, conDesc: execution of AfterUpdate caused by: System.ListException: Duplicate id in list: 0015g00001P75lDAAR Trigger.conDesc: line 25, column 1: [] Trigger.DescCon: line 22, column 1: [] Trigger.conDesc: line 25, column 1**

**Helper Class**

public class TriggerHelper {  
    public static Boolean isTriggerExecuted = false;  
}

**Trigger**

trigger conDesc on Contact (after update) {  
    if (TriggerHelper.isTriggerExecuted) {  
        return;  
    }

    TriggerHelper.isTriggerExecuted = true;

    Set<Id> conIdSet = new Set<Id>();

    // Collect Contact IDs where the description has changed  
    for (Contact conNew : Trigger.new) {  
        Contact conOld = Trigger.oldMap.get(conNew.Id);  
        if (conNew.Description != conOld.Description) {  
            conIdSet.add(conNew.Id);  
        }  
    }

    // Query Contacts with their Account IDs  
    List<Contact> conWithAcc = [SELECT Id, Description, AccountId FROM Contact WHERE Id IN :conIdSet];

    Map<Id, Account> accUpdateMap = new Map<Id, Account>();  
    for (Contact con : conWithAcc) {  
        if (con.AccountId != null) {  
            if (!accUpdateMap.containsKey(con.AccountId)) {  
                accUpdateMap.put(con.AccountId, new Account(Id = con.AccountId, Description = con.Description));  
            } else {  
                accUpdateMap.get(con.AccountId).Description = con.Description;  
            }  
        }  
    }

    // Update Accounts if there are any to update  
    if (!accUpdateMap.isEmpty()) {  
        update accUpdateMap.values();  
    }

    TriggerHelper.isTriggerExecuted = false;  
}

**Trigger my code but not optimise**

trigger conDesc on Contact (after Update) {  
    if(TriggerHelper.isTriggerExecuted){  
        return;  
    }  
      
    TriggerHelper.isTriggerExecuted = true;  
      
    if(Trigger.isAfter && Trigger.isUpdate){  
        Set<Id> ConId = new Set<Id>();  
        for(Contact connew: Trigger.new){  
            Contact conold = Trigger.oldMap.get(connew.id);  
            if(connew.Description != conold.Description ){  
              ConId.add(connew.Id);  
            }  
        }  
          
        List<Contact> conWithacc = [SELECT Id, Description, AccountId FROM Contact WHERE Id IN : ConId];  
      
        List<Account> accUpdate = new List<Account>();  
        for(Contact con: conWithacc){  
            if(con.AccountId != null){  
            Account acc = new Account();  
            acc.Id = con.AccountId;  
            acc.Description = con.Description;  
              
            accUpdate.add(acc);  
                System.debug(accUpdate);  
        }  
        }  
        if(!accUpdate.isEmpty()){  
            UPDATE accUpdate;  
        }  
    }  
      
    TriggerHelper.isTriggerExecuted =false;  
      
}

**Yes, there are several ways to handle recursive triggers in Salesforce. Here are a few common approaches:**

**1. Static Variables**

As we discussed, using a static variable in a helper class is a common method to prevent recursion. This ensures the trigger logic only runs once per transaction.

**2. Custom Settings or Custom Metadata**

You can use custom settings or custom metadata to track the state of the trigger execution. This approach is more persistent and can be useful if you need to maintain state across different transactions or contexts.

**Example with Custom Settings:**

1. **Create a Custom Setting: Create a custom setting called TriggerControl with a checkbox field isTriggerExecuted.**
2. **Trigger Code:**

trigger conDesc on Contact (after update) {  
    TriggerControl\_\_c control = TriggerControl\_\_c.getInstance('TriggerControl');  
    if (control.isTriggerExecuted\_\_c) {  
        return;  
    }  
  
    control.isTriggerExecuted\_\_c = true;  
    update control;  
  
    Set<Id> conIdSet = new Set<Id>();  
  
    for (Contact conNew : Trigger.new) {  
        Contact conOld = Trigger.oldMap.get(conNew.Id);  
        if (conNew.Description != conOld.Description) {  
            conIdSet.add(conNew.Id);  
        }  
    }  
  
    List<Contact> conWithAcc = [SELECT Id, Description, AccountId FROM Contact WHERE Id IN :conIdSet];  
  
    Map<Id, Account> accUpdateMap = new Map<Id, Account>();  
    for (Contact con : conWithAcc) {  
        if (con.AccountId != null) {  
            if (!accUpdateMap.containsKey(con.AccountId)) {  
                accUpdateMap.put(con.AccountId, new Account(Id = con.AccountId, Description = con.Description));  
            } else {  
                accUpdateMap.get(con.AccountId).Description = con.Description;  
            }  
        }  
    }  
  
    if (!accUpdateMap.isEmpty()) {  
        update accUpdateMap.values();  
    }  
  
    control.isTriggerExecuted\_\_c = false;  
    update control;  
}

**3. Handler Classes**

Using a handler class to manage your trigger logic can help you control the flow and prevent recursion. This approach also makes your code more modular and easier to maintain.

**Example with Handler Class:**

1. **Handler Class:**

public class ContactTriggerHandler {  
    public static Boolean isTriggerExecuted = false;  
  
    public static void afterUpdate(List<Contact> newContacts, Map<Id, Contact> oldMap) {  
        if (isTriggerExecuted) {  
            return;  
        }  
  
        isTriggerExecuted = true;  
  
        Set<Id> conIdSet = new Set<Id>();  
  
        for (Contact conNew : newContacts) {  
            Contact conOld = oldMap.get(conNew.Id);  
            if (conNew.Description != conOld.Description) {  
                conIdSet.add(conNew.Id);  
            }  
        }  
  
        List<Contact> conWithAcc = [SELECT Id, Description, AccountId FROM Contact WHERE Id IN :conIdSet];  
  
        Map<Id, Account> accUpdateMap = new Map<Id, Account>();  
        for (Contact con : conWithAcc) {  
            if (con.AccountId != null) {  
                if (!accUpdateMap.containsKey(con.AccountId)) {  
                    accUpdateMap.put(con.AccountId, new Account(Id = con.AccountId, Description = con.Description));  
                } else {  
                    accUpdateMap.get(con.AccountId).Description = con.Description;  
                }  
            }  
        }  
  
        if (!accUpdateMap.isEmpty()) {  
            update accUpdateMap.values();  
        }  
  
        isTriggerExecuted = false;  
    }  
}

1. **Trigger:**

trigger conDesc on Contact (after update) {  
    ContactTriggerHandler.afterUpdate(Trigger.new, Trigger.oldMap);  
}

**4. Platform Events**

For more complex scenarios, you can use Platform Events to handle updates asynchronously. This approach decouples the trigger logic and can help avoid recursion by processing updates in a separate transaction.

**Example with Platform Events:**

1. **Define a Platform Event: Create a platform event called ContactUpdateEvent.**
2. **Trigger to Publish Event:**

**trigger conDesc on Contact (after update) {**    List<ContactUpdateEvent\_\_e> events = new List<ContactUpdateEvent\_\_e>();  
  
    for (Contact conNew : Trigger.new) {  
        Contact conOld = Trigger.oldMap.get(conNew.Id);  
        if (conNew.Description != conOld.Description) {  
            ContactUpdateEvent\_\_e event = new ContactUpdateEvent\_\_e(  
                ContactId\_\_c = conNew.Id,  
                NewDescription\_\_c = conNew.Description  
            );  
            events.add(event);  
        }  
    }  
  
    if (!events.isEmpty()) {  
        EventBus.publish(events);  
    }  
}

1. **Event Trigger:**

trigger ContactUpdateEventTrigger on ContactUpdateEvent\_\_e (after insert) {  
    Set<Id> conIdSet = new Set<Id>();  
  
    for (ContactUpdateEvent\_\_e event : Trigger.new) {  
        conIdSet.add(event.ContactId\_\_c);  
    }  
  
    List<Contact> conWithAcc = [SELECT Id, Description, AccountId FROM Contact WHERE Id IN :conIdSet];  
  
    Map<Id, Account> accUpdateMap = new Map<Id, Account>();  
    for (Contact con : conWithAcc) {  
        if (con.AccountId != null) {  
            if (!accUpdateMap.containsKey(con.AccountId)) {  
                accUpdateMap.put(con.AccountId, new Account(Id = con.AccountId, Description = con.Description));  
            } else {  
                accUpdateMap.get(con.AccountId).Description = con.Description;  
            }  
        }  
    }  
  
    if (!accUpdateMap.isEmpty()) {  
        update accUpdateMap.values();  
    }  
}

**Trigger interview Questions**

**1. What is a trigger in Salesforce?**

Answer: A trigger is an Apex script that executes before or after specific data manipulation events, such as insert, update, delete, and undelete, on a particular Salesforce object. Triggers allow you to perform custom actions before or after changes to Salesforce records.

**2. What are the different types of triggers in Salesforce?**

Answer: There are two main types of triggers:

* **Before Triggers**: Used to update or validate record values before they are saved to the database.
* **After Triggers**: Used to access field values that are set by the system (such as record IDs) and to make changes in other records. After triggers are typically used when you need to perform operations that require the record to be committed to the database.

**3. Can you explain the trigger context variables?**

**Answer: Trigger context variables provide context about the trigger execution. Some common context variables include:**

* **Trigger.new**: A list of new versions of the sObject records.
* **Trigger.old**: A list of old versions of the sObject records.
* **Trigger.newMap**: A map of IDs to the new versions of the sObject records.
* **Trigger.oldMap**: A map of IDs to the old versions of the sObject records.
* **Trigger.isInsert, Trigger.isUpdate, Trigger.isDelete, Trigger.isUndelete: Boolean variables indicating the type of operation.**
* Trigger.isBefore, Trigger.isAfter: Boolean variables indicating the timing of the trigger.

**4. How do you prevent recursion in triggers?**

Answer: Recursion can be prevented using static variables, Platform Events, Handler Classes, Custom Settings or Custom Metadata.

**5. What are some best practices for writing triggers?**

**Answer:**

* One Trigger Per Object: Ensure there is only one trigger per object to avoid conflicts and manage logic more easily.
* Use Handler Classes: Delegate the logic to handler classes to keep the trigger code clean and maintainable.
* Bulkify Your Code: Ensure your trigger can handle bulk operations by processing records in collections.
* Avoid SOQL and DML in Loops: Minimize the number of SOQL queries and DML operations to avoid hitting governor limits.
* Use Context Variables: Leverage context variables to write efficient and effective trigger logic.

**6. What are governor limits and how do they affect triggers?**

Answer: Governor limits are runtime limits enforced by the Salesforce platform to ensure efficient use of resources. They include limits on the number of SOQL queries, DML statements, CPU time, heap size, and more. Triggers must be written to respect these limits to avoid runtime exceptions. Bulkifying code and avoiding SOQL/DML operations inside loops are key strategies to stay within these limits.

**Flow Interview Questions**

**Salesforce Flow is a powerful tool for automating business processes without writing code. Here are some common interview questions related to Salesforce Flow, along with tips on how to answer them:**

**Basic Questions**

1. **What is Salesforce Flow?**

Answer: Salesforce Flow is a declarative automation tool that allows users to create complex business processes using a visual interface. It includes two main types: Screen Flows (for user interactions) and Autolaunched Flows (for background processes).

1. **What are the different types of Flows in Salesforce?**

**Answer: The main types of Flows are:**

* + Screen Flows: Require user interaction and can be used to guide users through a series of screens.
  + Autolaunched Flows: Run in the background and can be triggered by events, such as record changes or platform events.
  + Scheduled Flows: Run at specified times and intervals.
  + Record-Triggered Flows: Automatically run when a record is created, updated, or deleted.

1. **How do you create a Flow in Salesforce?**

Answer: To create a Flow, navigate to Setup -> Process Automation -> Flows. Click “New Flow” and choose the type of Flow you want to create. Use the Flow Builder to drag and drop elements, configure their properties, and connect them to define the process.

**Scenario-Based Questions**

1. **Scenario: You need to create a Flow that updates a field on the Account object when a related Contact is created or updated. How would you implement this?**

Answer:

* + Trigger: Use a Record-Triggered Flow on the Contact object.
  + Criteria: Set the criteria to run when a Contact is created or updated.
  + Action: Use an Update Records element to update the related Account field.

1. **Scenario: How would you use a Flow to send an email notification when an Opportunity’s Stage is changed to “Closed Won”?**

Answer:

* + Trigger: Use a Record-Triggered Flow on the Opportunity object.
  + Criteria: Set the criteria to run when the Stage field is changed to “Closed Won”.
  + Action: Use a Send Email action to send the notification.

1. **Scenario: You need to create a Flow that collects user input through a series of screens and then creates a new Case record. How would you implement this?**

**Answer:**

* + Flow Type: Use a Screen Flow.
  + Screens: Add Screen elements to collect user input.
  + Create Record: Use a Create Records element to create the new Case record with the collected input.

**Advanced Questions**

1. **How do you handle errors in Salesforce Flow?**

Answer: Error handling in Flows can be managed by configuring Fault Paths on elements that can fail, such as data operations. You can use Fault Paths to log errors, send notifications, or take alternative actions when an error occurs.

1. **What are Flow Variables and how are they used?**

Answer: Flow Variables are used to store data that can be referenced and manipulated throughout the Flow. They can be used to pass data between elements, store user input, or hold values for calculations.

1. **How do you debug a Flow in Salesforce?**

Answer: You can debug a Flow by using the Debug button in the Flow Builder. This allows you to run the Flow step-by-step, view the values of variables, and see the path taken through the Flow. Additionally, you can use debug logs to capture detailed information about Flow execution.

1. **What are Flow Collections and how are they used?**

Answer: Flow Collections are variables that can store multiple records or values. They are used to handle bulk data operations, such as querying multiple records, looping through records, and performing actions on each record in the collection.

**Practical Questions**

1. **How would you use a Flow to automate the approval process for a custom object?**

Answer:

* + Flow Type: Use an Autolaunched Flow.
  + Trigger: Use a Record-Triggered Flow or a Process Builder to start the Flow when a record meets certain criteria.
  + Approval Process: Use the Submit for Approval action to submit the record for approval.

1. **Can you explain how to use the Loop element in a Flow?**

Answer: The Loop element is used to iterate over a collection of records or values. You define the collection to loop through and specify the actions to take for each item in the collection. Inside the loop, you can use Assignment elements to modify variables and Decision elements to branch logic.

**Apex Interview Questions**

**Basic Questions**

1. **What is Apex?**

Answer: Apex is a strongly-typed, object-oriented programming language that allows developers to execute flow and transaction control statements on the Salesforce platform server in conjunction with calls to the API.

1. **What are the different types of collections in Apex?**

Answer: The main types of collections in Apex are Lists, Sets, and Maps.

1. **What is the difference between Trigger.new and Trigger.old?**

Answer: Trigger.new contains the new versions of the sObject records, while Trigger.old contains the old versions of the sObject records before they were updated.

1. **What are governor limits in Salesforce?**

Answer: Governor limits are runtime limits enforced by the Salesforce platform to ensure efficient use of resources. They include limits on the number of SOQL queries, DML statements, CPU time, heap size, and more.

1. **What is a SOQL query?**

Answer: SOQL (Salesforce Object Query Language) is used to query data from Salesforce objects. It is similar to SQL but specifically designed for Salesforce data.

**Intermediate Questions**

1. **How do you handle exceptions in Apex?**

Answer: Exceptions in Apex are handled using try-catch blocks. You can catch specific exceptions or use a generic catch block to handle any exception.

1. **What is a future method in Apex?**

Answer: A future method is an asynchronous method that runs in the background. It is used for operations that need to be performed asynchronously, such as callouts to external services.

1. **What is the difference between Database.insert and insert?**

Answer: Database.insert allows for partial success and returns a result object that contains success and error information for each record. The insert statement does not allow for partial success and will throw an exception if any record fails.

1. **What is a batch Apex?**

Answer: Batch Apex is used to process large volumes of records asynchronously. It allows you to break down large jobs into smaller chunks and process them separately.

1. **What is the purpose of the @isTest annotation?**

Answer: The @isTest annotation is used to define test classes and test methods in Apex. It indicates that the class or method is a test and should not be included in the organization’s code coverage calculations.

**Advanced Questions**

1. **How do you prevent recursion in triggers?**

Answer: Recursion in triggers can be prevented using static variables in a helper class to ensure the trigger logic only runs once per transaction.

1. **What is the difference between before and after triggers?**

Answer: Before triggers are used to update or validate record values before they are saved to the database. After triggers are used to access field values that are set by the system and to make changes in other records.

1. **What are custom metadata types and how are they used in Apex?**

Answer: Custom metadata types are similar to custom objects but are used to define application metadata. They can be used in Apex to store configuration data that can be deployed between environments.

1. **What is the @InvocableMethod annotation?**

Answer: The @InvocableMethod annotation is used to define methods that can be called from a Flow or Process Builder. It allows you to create reusable Apex code that can be invoked from declarative tools.

1. **How do you perform callouts to external services in Apex?**

Answer: Callouts to external services are performed using the Http and HttpRequest classes in Apex. You can make GET, POST, PUT, and DELETE requests to external services.

**Practical Questions**

1. **How do you write a test class for an Apex trigger?**

Answer: A test class for an Apex trigger should create test data, execute the trigger, and use assertions to verify the expected outcomes. It should cover all possible scenarios, including positive and negative cases.

1. **What is the purpose of the @TestSetup annotation?**

Answer: The @TestSetup annotation is used to define a method that sets up test data that is shared across multiple test methods in a test class. It helps reduce code duplication and improves test performance.

1. **How do you use the Database.SaveResult class?**

Answer: The Database.SaveResult class is used to handle the results of DML operations performed using the Database class. It contains information about whether the operation was successful and any errors that occurred.

1. **What is the System.runAs method used for?**

Answer: The System.runAs method is used to execute code in the context of a different user. It is commonly used in test methods to test functionality that depends on user permissions.

1. **How do you optimize SOQL queries to avoid hitting governor limits?**

Answer: To optimize SOQL queries, you can use selective filters, limit the number of records returned, use indexed fields, and avoid querying unnecessary fields. Additionally, you can use query planning tools to analyze and optimize your queries.

**LWC Interview Question**

**Basic Questions**

1. **What is Lightning Web Components (LWC)?**

Answer: LWC is a modern JavaScript framework developed by Salesforce for building web applications on the Salesforce platform. It leverages web standards and provides a lightweight, performance-optimized way to create reusable components.

1. **What are the key differences between Aura Components and Lightning Web Components?**

Answer: LWC uses modern web standards and is more lightweight and performance-optimized compared to Aura Components. LWC leverages native browser APIs, while Aura relies on a proprietary framework.

1. **How do you create a Lightning Web Component?**

Answer: You can create an LWC using the Salesforce CLI by running the command sfdx force:lightning:component:create --type lwc --componentname <componentName> --outputdir <directory>.

1. **What is the role of the @api decorator in LWC?**

Answer: The @api decorator is used to expose public properties and methods of a component, making them accessible to parent components.

1. **How do you handle events in LWC?**

Answer: Events in LWC are handled using the standard DOM event model. You can create and dispatch events using the CustomEvent constructor and handle them using event listeners.

**Intermediate Questions**

1. **What is the @track decorator used for in LWC?**

Answer: The @track decorator is used to make a property reactive, meaning changes to the property will automatically update the component’s UI.

1. **How do you communicate between components in LWC?**

Answer: Communication between components can be achieved using public properties, custom events, and the Lightning Message Service (LMS) for pub-sub communication.

1. **What is the Lightning Data Service (LDS) and how is it used in LWC?**

Answer: LDS is a service that provides access to Salesforce data without the need for Apex controllers. It simplifies data access and ensures data consistency. In LWC, you can use LDS via the lightning-record-form, lightning-record-view-form, and lightning-record-edit-form components.

1. **How do you use the wire service in LWC?**

Answer: The wire service is used to connect a component to Salesforce data. You can use the @wire decorator to call Apex methods or standard Salesforce APIs and automatically refresh the component when the data changes.

1. **What are slots in LWC and how are they used?**

Answer: Slots are placeholders in a component’s template where content from a parent component can be inserted. They allow for flexible and reusable component designs.

**Advanced Questions**

1. **How do you handle lifecycle hooks in LWC?**

Answer: LWC provides several lifecycle hooks, such as connectedCallback, disconnectedCallback, renderedCallback, and errorCallback, which allow you to execute code at specific points in a component’s lifecycle.

1. **What is the @wire decorator and how does it differ from imperative Apex calls?**

Answer: The @wire decorator is used for reactive data binding, automatically updating the component when the data changes. Imperative Apex calls are made using JavaScript functions and require manual handling of data updates.

1. **How do you use CSS in LWC?**

Answer: CSS in LWC is scoped to the component by default, meaning styles defined in a component’s CSS file only apply to that component. You can use standard CSS, SLDS (Salesforce Lightning Design System) classes, and custom styles.

1. **What are the best practices for optimizing performance in LWC?**

Answer: Best practices include minimizing DOM updates, using the @wire decorator for reactive data binding, leveraging LDS, lazy loading components, and avoiding large JavaScript bundles.

1. **How do you handle form validation in LWC?**

Answer: Form validation can be handled using standard HTML5 validation attributes, custom validation logic in JavaScript, and leveraging the lightning-input component’s validation methods.

**Practical Questions**

1. **How do you deploy an LWC to a Salesforce org?**

Answer: You can deploy an LWC to a Salesforce org using the Salesforce CLI with the command sfdx force:source:deploy -p <path-to-component> or by using change sets in the Salesforce UI.

1. **How do you debug an LWC?**

Answer: Debugging an LWC can be done using browser developer tools to inspect the DOM, view console logs, and set breakpoints in the JavaScript code. Salesforce also provides the Lightning Web Components Developer Guide for troubleshooting.

1. **What is the Lightning Message Service (LMS) and how is it used in LWC?**

Answer: LMS is a service that enables communication between Lightning components, Visualforce pages, and Aura components using a publish-subscribe model. It is used to share data and events across the Salesforce UI.

1. **How do you handle asynchronous operations in LWC?**

Answer: Asynchronous operations in LWC are handled using JavaScript Promises, async/await syntax, and the @wire decorator for reactive data binding.

1. **How do you test an LWC?**

Answer: Testing an LWC can be done using the Jest framework for unit tests. Salesforce provides a Jest testing framework specifically for LWC, which allows you to write and run tests for your components.

**Omniscript Interview Question**

**Basic Questions**

1. **What is OmniScript in Salesforce?**

Answer: OmniScript is a declarative tool in Salesforce that allows users to create guided interactions and processes. It helps build complex, multi-step forms and workflows without writing code, providing a user-friendly interface for data collection and process automation.

1. **What are the key components of an OmniScript?**

**Answer: The key components of an OmniScript include:**

* + Steps: Define the stages of the interaction.
  + Elements: Individual building blocks like inputs, actions, and display elements.
  + Groups: Logical grouping of elements.
  + Actions: Perform operations like data retrieval, data updates, and integrations.

1. **How do you create an OmniScript?**

Answer: To create an OmniScript, navigate to the OmniStudio app in Salesforce, select OmniScripts, and click “New”. Define the script’s properties and use the OmniScript Designer to drag and drop elements to build the script.

**Intermediate Questions**

1. **What is the purpose of the DataRaptor in OmniScript?**

Answer: DataRaptor is a tool used within OmniScript to extract, transform, and load data. It allows you to retrieve data from Salesforce, transform it as needed, and save it back to Salesforce or other systems.

1. **How do you handle conditional logic in OmniScript?**

Answer: Conditional logic in OmniScript can be handled using the Conditional View property on elements, which allows you to show or hide elements based on specific conditions. You can also use Decision elements to branch the flow based on certain criteria.

1. **What are Integration Procedures in OmniScript?**

Answer: Integration Procedures are server-side processes that allow you to perform complex data operations and integrations with external systems. They are used to call APIs, perform data transformations, and handle large volumes of data efficiently.

**Advanced Questions**

1. **How do you debug an OmniScript?**

Answer: Debugging an OmniScript can be done using the OmniScript Debugger, which allows you to test the script step-by-step, view the data at each step, and identify any issues. You can also use console logs and Salesforce debug logs for more detailed troubleshooting.

1. **What is the difference between a Step and a Block in OmniScript?**

Answer: A Step represents a stage in the OmniScript flow, typically corresponding to a page or screen in the user interface. A Block is a container within a Step that groups related elements together, allowing for better organization and layout control.

1. **How do you handle data validation in OmniScript?**

Answer: Data validation in OmniScript can be handled using the Validation Rules property on input elements, which allows you to define custom validation logic. You can also use Decision elements to enforce validation rules and guide the user through the correct process.

**Practical Questions**

1. **Scenario: You need to create an OmniScript that collects customer information, validates the data, and creates a new Case record in Salesforce. How would you implement this?**

**Answer:**

* + Steps: Create steps for data collection, validation, and confirmation.
  + Elements: Use input elements to collect customer information, decision elements for validation, and actions to create the Case record.
  + DataRaptor: Use a DataRaptor Load to save the data to the Case object.
  + Integration Procedures: If needed, use integration procedures to handle complex data operations or external system integrations.

**DataMapper Interview Question**

**Basic Questions**

1. **What is DataRaptor in Salesforce?**

Answer: DataRaptor is a declarative tool in Salesforce OmniStudio that allows users to extract, transform, and load data. It is used to retrieve data from Salesforce, transform it as needed, and save it back to Salesforce or other systems.

1. **What are the different types of DataRaptors?**

**Answer: The main types of DataRaptors are:**

* + DataRaptor Extract: Retrieves data from Salesforce.
  + DataRaptor Load: Saves data to Salesforce.
  + DataRaptor Transform: Transforms data from one format to another.
  + DataRaptor Turbo Extract: A more efficient version of DataRaptor Extract for retrieving data.

1. **How do you create a DataRaptor in Salesforce?**

Answer: To create a DataRaptor, navigate to the OmniStudio app in Salesforce, select DataRaptors, and click “New”. Choose the type of DataRaptor you want to create and configure its properties using the DataRaptor Designer.

**Intermediate Questions**

1. **What is the purpose of the DataRaptor Extract?**

Answer: DataRaptor Extract is used to retrieve data from Salesforce objects. It allows you to specify the fields and conditions for the data you want to extract.

1. **How do you handle data transformations in DataRaptor?**

Answer: Data transformations in DataRaptor can be handled using formulas, mappings, and functions within the DataRaptor Designer. You can define how data should be transformed before it is loaded or displayed.

1. **What is the difference between DataRaptor Extract and DataRaptor Turbo Extract?**

Answer: DataRaptor Turbo Extract is optimized for performance and is used for retrieving large volumes of data quickly. It has fewer configuration options compared to DataRaptor Extract but is more efficient for simple data retrieval tasks.

**Advanced Questions**

1. **How do you handle complex data mappings in DataRaptor?**

Answer: Complex data mappings in DataRaptor can be handled using nested mappings, formulas, and functions. You can define detailed mappings to transform data from source to target formats.

1. **What are some best practices for optimizing DataRaptor performance?**

**Answer: Best practices for optimizing DataRaptor performance include:**

* + Using DataRaptor Turbo Extract for simple data retrieval.
  + Minimizing the number of fields and conditions in your DataRaptor.
  + Using efficient data transformations and avoiding unnecessary complexity.
  + Testing and tuning your DataRaptor configurations for optimal performance.

1. **How do you debug a DataRaptor?**

Answer: Debugging a DataRaptor can be done using the DataRaptor Designer’s preview feature, which allows you to test your configurations and view the output. You can also use Salesforce debug logs and error messages to troubleshoot issues.

**Practical Questions**

1. **Scenario: You need to create a DataRaptor that retrieves account data and transforms it into a specific format for an external system. How would you implement this?**

**Answer:**

* + DataRaptor Extract: Create a DataRaptor Extract to retrieve the required account data from Salesforce.
  + DataRaptor Transform: Use a DataRaptor Transform to convert the data into the required format for the external system.
  + Mappings and Formulas: Define the necessary mappings and formulas to transform the data as needed.
  + Testing: Use the preview feature to test the DataRaptor and ensure the output meets the requirements.

**Integration Procedure Interview Question**

**Basic Questions**

1. **What is an Integration Procedure in Salesforce OmniStudio?**

Answer: An Integration Procedure is a declarative tool in Salesforce OmniStudio used to perform server-side processing, such as data retrieval, transformation, and integration with external systems. It allows for complex data operations without writing code.

1. **What are the main components of an Integration Procedure?**

**Answer: The main components include:**

* + Elements: Building blocks like DataRaptor, HTTP Action, and Apex Action.
  + Steps: Logical groupings of elements that define the sequence of operations.
  + Conditions: Criteria that determine when certain steps or elements should be executed.

1. **How do you create an Integration Procedure?**

Answer: To create an Integration Procedure, navigate to the OmniStudio app in Salesforce, select Integration Procedures, and click “New”. Use the Integration Procedure Designer to add and configure elements and steps.

**Intermediate Questions**

1. **What is the purpose of the DataRaptor Extract element in an Integration Procedure?**

Answer: The DataRaptor Extract element is used to retrieve data from Salesforce objects. It allows you to specify the fields and conditions for the data you want to extract.

1. **How do you handle conditional logic in an Integration Procedure?**

Answer: Conditional logic can be handled using the Conditional View property on elements and steps, which allows you to execute certain operations based on specific conditions.

1. **What is the difference between a DataRaptor Extract and an HTTP Action in an Integration Procedure?**

Answer: A DataRaptor Extract retrieves data from Salesforce objects, while an HTTP Action is used to make callouts to external systems via REST or SOAP APIs.

**Advanced Questions**

1. **How do you optimize the performance of an Integration Procedure?**

Answer: Performance can be optimized by minimizing the number of elements and steps, using efficient data transformations, leveraging caching, and avoiding unnecessary data retrieval or processing.

1. **What are the best practices for error handling in an Integration Procedure?**

Answer: Best practices include using the Error Handling element to capture and log errors, defining clear error messages, and implementing retry logic for transient errors.

1. **How do you debug an Integration Procedure?**

Answer: Debugging can be done using the Integration Procedure Debugger, which allows you to test the procedure step-by-step, view the data at each step, and identify any issues. You can also use Salesforce debug logs and error messages for more detailed troubleshooting.

**Practical Questions**

1. **Scenario: You need to create an Integration Procedure that retrieves account data from Salesforce and updates it in an external system. How would you implement this?**

**Answer:**

* + DataRaptor Extract: Use a DataRaptor Extract to retrieve the account data from Salesforce.
  + HTTP Action: Use an HTTP Action to send the data to the external system.
  + Transformations: Apply any necessary data transformations before sending the data.

1. **Scenario: How would you use an Integration Procedure to handle a complex data transformation involving multiple Salesforce objects?**

**Answer:**

* + DataRaptor Extract: Use multiple DataRaptor Extract elements to retrieve data from the different objects.
  + DataRaptor Transform: Use DataRaptor Transform elements to perform the necessary data transformations.
  + Steps: Organize the elements into logical steps to ensure the correct sequence of operations.

1. **What is the purpose of the Cache Block element in an Integration Procedure?**

Answer: The Cache Block element is used to store and retrieve data in a cache, improving performance by reducing the need for repeated data retrieval or processing.

1. **How do you handle pagination in an Integration Procedure?**

Answer: Pagination can be handled using the Pagination element, which allows you to retrieve and process data in chunks, reducing the load on the system and improving performance.

1. **What is the difference between a Response Action and a Response Transform in an Integration Procedure?**

Answer: A Response Action defines the structure of the response returned by the Integration Procedure, while a Response Transform applies transformations to the data before it is returned.

1. **How do you use the Loop Block element in an Integration Procedure?**

Answer: The Loop Block element is used to iterate over a collection of data, allowing you to perform operations on each item in the collection.

1. **Scenario: You need to integrate Salesforce with an external payment gateway. How would you use an Integration Procedure to achieve this?**

**Answer:**

* + HTTP Action: Use an HTTP Action to send payment data to the external gateway.
  + DataRaptor Extract: Retrieve necessary data from Salesforce using a DataRaptor Extract.
  + Error Handling: Implement error handling to manage any issues with the payment gateway.

1. **How do you use the Set Values element in an Integration Procedure?**

Answer: The Set Values element is used to assign values to variables within the Integration Procedure, allowing you to manipulate data and control the flow of the procedure.

1. **What is the purpose of the Response Action element in an Integration Procedure?**

Answer: The Response Action element defines the structure and content of the response returned by the Integration Procedure, ensuring that the output meets the requirements of the calling process.

1. **How do you handle authentication for external API calls in an Integration Procedure?**

Answer: Authentication can be handled using the HTTP Action element’s authentication settings, such as OAuth, Basic Authentication, or custom headers.

1. **Scenario: You need to create an Integration Procedure that processes a large volume of data from an external system and updates Salesforce records. How would you implement this?**

**Answer:**

* + HTTP Action: Use an HTTP Action to retrieve the data from the external system.
  + Loop Block: Use a Loop Block to process each item in the data collection.
  + DataRaptor Load: Use a DataRaptor Load to update the Salesforce records.
  + Error Handling: Implement error handling to manage any issues during the data processing.

**Service Cloud Interview Question**

**Basic Questions**

1. **What is Salesforce Service Cloud?**

Answer: Salesforce Service Cloud is a customer service and support platform that helps businesses manage customer interactions and support cases. It provides tools for case management, knowledge base, customer self-service, and more.

1. **What are the key features of Service Cloud?**

Answer: Key features include Case Management, Knowledge Base, Service Console, Omni-Channel Routing, Live Agent Chat, Service Analytics, and Customer Communities.

1. **How does Case Management work in Service Cloud?**

Answer: Case Management allows you to track and manage customer issues. Cases can be created from various channels (email, phone, web), assigned to agents, and tracked through to resolution.

1. **What is the Service Console?**

Answer: The Service Console is a unified interface for support agents that provides a 360-degree view of customer interactions, allowing agents to manage cases, access customer information, and collaborate with other team members efficiently.

**Intermediate Questions**

1. **What is Omni-Channel Routing in Service Cloud?**

Answer: Omni-Channel Routing automatically routes cases, leads, and other work items to the most appropriate and available agents based on predefined criteria, ensuring efficient workload distribution.

1. **How do you set up Email-to-Case in Service Cloud?**

Answer: Email-to-Case allows customers to create cases by sending emails. It can be set up by configuring email services in Salesforce, defining email addresses, and setting up routing rules to create cases from incoming emails.

1. **What is a Knowledge Base and how is it used in Service Cloud?**

Answer: A Knowledge Base is a repository of articles and information that agents and customers can use to find answers to common questions and issues. It helps improve self-service and agent efficiency.

1. **How do you use Macros in the Service Console?**

Answer: Macros automate repetitive tasks in the Service Console. Agents can create and run macros to perform multiple actions with a single click, such as updating case fields, sending emails, and logging activities.

1. **What is Live Agent Chat and how is it configured?**

Answer: Live Agent Chat allows customers to chat with support agents in real-time. It can be configured by setting up chat buttons, deploying chat to your website, and configuring agent availability and routing rules.

**Advanced Questions**

1. **How do you implement a Customer Community in Service Cloud?**

Answer: A Customer Community is a self-service portal where customers can find answers, submit cases, and interact with other customers. It can be implemented by creating a community in Salesforce, configuring community settings, and customizing the community with branding and content.

1. **What is Service Analytics and how is it used?**

Answer: Service Analytics provides insights into service performance using pre-built dashboards and reports. It helps managers track key metrics like case resolution times, agent performance, and customer satisfaction.

1. **How do you integrate Service Cloud with external systems?**

Answer: Integration can be achieved using Salesforce APIs, middleware tools like MuleSoft, and custom Apex code. Common integrations include CRM systems, ERP systems, and third-party support tools.

1. **What are Entitlements and Milestones in Service Cloud?**

Answer: Entitlements define the level of support a customer is entitled to, while Milestones are key steps in the support process that must be completed within specified timeframes. They help ensure SLAs are met.

1. **How do you use the Case Feed in Service Cloud?**

Answer: The Case Feed provides a chronological view of all interactions and updates related to a case. It helps agents stay informed about the case history and collaborate more effectively.

**Practical Questions**

1. **Scenario: You need to set up a process to automatically escalate cases that are not resolved within 24 hours. How would you implement this?**

Answer: Use Escalation Rules to define criteria for escalating cases. Set up a rule to check if a case is not resolved within 24 hours and specify the actions to take, such as reassigning the case or notifying a manager.

1. **Scenario: How would you configure a Knowledge Base to improve customer self-service?**

Answer: Create and organize knowledge articles, set up data categories for easy navigation, enable article ratings and feedback, and configure Knowledge Search to help customers find relevant articles quickly.

1. **Scenario: You need to provide support agents with a unified view of customer interactions across multiple channels. How would you achieve this?**

Answer: Use the Service Console to provide a unified interface. Configure Omni-Channel Routing to manage interactions from different channels and integrate with other systems to pull in customer data.

1. **Scenario: How would you implement a chatbot to handle common customer inquiries?**

Answer: Use Einstein Bots to create a chatbot. Define the bot’s dialog, configure it to handle common inquiries, integrate it with the Knowledge Base for answers, and deploy it on your website or community.

1. **Scenario: How do you ensure that high-priority cases are handled promptly by the support team?**

Answer: Use Case Assignment Rules to route high-priority cases to senior agents or specialized teams. Set up Escalation Rules to ensure timely follow-up and use Omni-Channel Routing to prioritize high-priority cases.

1. **Scenario: How would you measure and improve customer satisfaction in Service Cloud?**

Answer: Use Service Analytics to track key metrics like case resolution times and customer feedback. Implement surveys to gather customer satisfaction data and use the insights to identify areas for improvement.

**Health Cloud**

1. **What is Salesforce Health Cloud?**
   * Salesforce Health Cloud is a cloud-based CRM platform designed specifically for the healthcare industry. It streamlines patient engagement, care coordination, and overall healthcare operations.
2. **How does Health Cloud differ from traditional Salesforce CRM?**
   * Health Cloud includes industry-specific features like care plans, social determinants of health tracking, and compliance with HIPAA, which are not present in the traditional Salesforce CRM[[1]](https://www.saasguru.co/salesforce-health-cloud-interview-questions-and-answers/).
3. **What are the core functionalities of Health Cloud?**
   * Core functionalities include patient management, care plan creation and management, appointment scheduling, secure communication, and integration with Electronic Health Records (EHR) systems[[1]](https://www.saasguru.co/salesforce-health-cloud-interview-questions-and-answers/).
4. **Explain the concept of a “Person Account” in Health Cloud.**
   * Person Accounts combine the functions of an account and a contact into one, storing all patient information in a single, unified record[[1]](https://www.saasguru.co/salesforce-health-cloud-interview-questions-and-answers/).
5. **What are Care Plans in Health Cloud?**
   * Care Plans are personalized healthcare roadmaps outlining specific health conditions, desired outcomes, and actionable steps to achieve those goals[[1]](https://www.saasguru.co/salesforce-health-cloud-interview-questions-and-answers/).
6. **How can users manage Care Plans within Health Cloud?**
   * Users can create, track, and update Care Plans by specifying health concerns, defining goals, and outlining tasks needed to achieve those goals[[1]](https://www.saasguru.co/salesforce-health-cloud-interview-questions-and-answers/).
7. **What is Health Cloud HL7 Integration?**
   * HL7 Integration allows seamless data transfer between Health Cloud and EHR systems, facilitating electronic exchange of healthcare information[[1]](https://www.saasguru.co/salesforce-health-cloud-interview-questions-and-answers/).
8. **What are the benefits of using Health Cloud Utilization Management?**
   * Benefits include reduced overutilization, adherence to evidence-based care, lower medical expenses, optimized resource allocation, faster approvals, and improved transparency[[1]](https://www.saasguru.co/salesforce-health-cloud-interview-questions-and-answers/).
9. **Briefly explain FHIR (Fast Healthcare Interoperability Resources).**
   * FHIR is a standard for exchanging healthcare information electronically, enabling different EHR systems to communicate seamlessly[[1]](https://www.saasguru.co/salesforce-health-cloud-interview-questions-and-answers/).
10. **How does Salesforce Health Cloud support Patient Engagement?**
    * Health Cloud supports patient engagement through secure messaging, appointment reminders, and self-service portals, empowering patients to actively participate in their care[[1]](https://www.saasguru.co/salesforce-health-cloud-interview-questions-and-answers/).
11. **What is the role of “Social Determinants of Health” in Health Cloud?**
    * Social determinants like income, housing, and social support are tracked in Health Cloud to provide more holistic patient care[[1]](https://www.saasguru.co/salesforce-health-cloud-interview-questions-and-answers/).
12. **Describe the concept of “Patient Reported Outcomes” (PROs) in Health Cloud.**
    * PROs are self-reported measures of a patient’s health status, symptoms, or quality of life, collected through surveys or questionnaires[[1]](https://www.saasguru.co/salesforce-health-cloud-interview-questions-and-answers/).
13. **How does Health Cloud ensure data security and compliance with HIPAA?**
    * Health Cloud ensures data security and HIPAA compliance through encryption, granular access controls, and regular audits[[1]](https://www.saasguru.co/salesforce-health-cloud-interview-questions-and-answers/).
14. **How would you customize Salesforce Health Cloud to meet the specific needs of a healthcare organization?**
    * Customization involves configuring the platform to align with the organization’s workflows, processes, and data requirements, including creating custom objects, fields, and workflows[[1]](https://www.saasguru.co/salesforce-health-cloud-interview-questions-and-answers/).
15. **Can you discuss the role of telehealth in Salesforce Health Cloud?**
    * Telehealth capabilities in Health Cloud enable remote patient monitoring, virtual consultations, and telemedicine services, enhancing access to care[[1]](https://www.saasguru.co/salesforce-health-cloud-interview-questions-and-answers/).
16. **What are some common challenges organizations face during Salesforce Health Cloud implementation, and how would you address them?**
    * Common challenges include data migration issues, user adoption hurdles, and ensuring compliance with regulatory requirements. Addressing these requires meticulous planning, stakeholder engagement, and comprehensive training programs[[1]](https://www.saasguru.co/salesforce-health-cloud-interview-questions-and-answers/).
17. **How does Salesforce Health Cloud support population health management initiatives?**
    * Health Cloud provides insights into patient populations through analytics and reporting tools, enabling targeted interventions to improve population health[[1]](https://www.saasguru.co/salesforce-health-cloud-interview-questions-and-answers/).
18. **What role does data integration play in Salesforce Health Cloud?**
    * Data integration allows Health Cloud to access and synchronize patient information from various sources, ensuring a unified view of patient data[[1]](https://www.saasguru.co/salesforce-health-cloud-interview-questions-and-answers/).
19. **How do you stay updated with the latest developments in Salesforce Health Cloud?**
    * Staying updated involves participating in Salesforce community forums, attending webinars and conferences, and completing relevant training modules on platforms like Salesforce Trailhead[[1]](https://www.saasguru.co/salesforce-health-cloud-interview-questions-and-answers/).
20. **What motivated you to pursue a career in Salesforce Health Cloud?**
    * Share your personal motivation, whether it’s a passion for healthcare technology, a desire to improve patient outcomes, or a fascination with the intersection of technology and healthcare[[1]](https://www.saasguru.co/salesforce-health-cloud-interview-questions-and-answers/).

**Asynchronous process Interview Question**

**Basic Questions**

1. **What is asynchronous processing in Salesforce?**

Answer: Asynchronous processing allows operations to run in the background without blocking the main execution thread. It is used for long-running or resource-intensive tasks that do not need to be completed immediately.

1. **What are the different types of asynchronous processing in Salesforce?**

**Answer: The main types include:**

* + Future Methods
  + Batch Apex
  + Queueable Apex
  + Scheduled Apex
  + Platform Events

1. **What is a future method in Salesforce?**

Answer: A future method is an asynchronous method annotated with @future. It runs in the background and is used for operations like callouts to external services or processing large datasets.

1. **How do you define a future method?**

Answer: A future method is defined using the @future annotation. It must be static and can only accept primitive data types or collections of primitive data types as parameters.

**Intermediate Questions**

1. **What is Batch Apex and when would you use it?**

Answer: Batch Apex is used to process large volumes of records asynchronously. It breaks down the job into smaller chunks and processes them separately, making it suitable for operations that need to handle large datasets.

1. **How do you implement Batch Apex?**

Answer: Implement Batch Apex by creating a class that implements the Database.Batchable interface. Define the start, execute, and finish methods to specify the batch processing logic.

1. **What is Queueable Apex and how does it differ from Future Methods?**

Answer: Queueable Apex is similar to future methods but provides more flexibility. It allows for complex job chaining and supports non-primitive types as parameters. It is implemented by creating a class that implements the Queueable interface.

1. **How do you chain Queueable jobs?**

Answer: Chain Queueable jobs by enqueuing another Queueable job from within the execute method of the current job using the System.enqueueJob method.

1. **What is Scheduled Apex and how is it used?**

Answer: Scheduled Apex allows you to schedule Apex classes to run at specific times. It is used for recurring tasks like nightly data processing or periodic updates. Implement it by creating a class that implements the Schedulable interface and using the System.schedule method to schedule it.

**Advanced Questions**

1. **What are Platform Events and how are they used in asynchronous processing?**

Answer: Platform Events are used for event-driven architecture in Salesforce. They allow you to publish and subscribe to events, enabling asynchronous communication between different parts of the system or with external systems.

1. **How do you publish and subscribe to Platform Events?**

Answer: Publish Platform Events using the EventBus.publish method. Subscribe to Platform Events using triggers, Process Builder, or Flow.

1. **What are the governor limits for asynchronous processing in Salesforce?**

Answer: Governor limits for asynchronous processing include limits on the number of future method invocations, batch jobs, queueable jobs, and scheduled jobs that can be executed per 24-hour period.

1. **How do you handle errors in asynchronous processing?**

Answer: Handle errors using try-catch blocks within the asynchronous methods. For Batch Apex, implement the Database.Stateful interface to maintain state across transactions and log errors in the finish method.

**Practical Questions**

1. **Scenario: You need to process a large number of records and update a field on each record. How would you implement this using Batch Apex?**

**Answer:**

* + Start Method: Query the records to be processed.
  + Execute Method: Update the field on each record in batches.
  + Finish Method: Perform any final operations or logging.

1. **Scenario: How would you make a callout to an external service from a future method?**

Answer: Define a future method with the @future(callout=true) annotation. Implement the callout logic within the method.

1. **Scenario: You need to schedule a job to run every night at midnight. How would you implement this using Scheduled Apex?**

**Answer:**

* + Schedulable Class: Create a class that implements the Schedulable interface.
  + Schedule Method: Use the System.schedule method to schedule the job to run at midnight.

1. **Scenario: How would you handle a scenario where a Queueable job needs to process records in multiple steps?**

Answer: Chain Queueable jobs by enqueuing the next job from within the execute method of the current job.

1. **Scenario: How would you use Platform Events to notify external systems of changes in Salesforce data?**

Answer: Publish a Platform Event whenever the data changes. External systems can subscribe to the event using the Salesforce Streaming API or other integration tools.

1. **Scenario: How do you ensure that a batch job does not exceed governor limits when processing a large dataset?**

Answer: Implement efficient query and processing logic, use the Database.QueryLocator for large queries, and handle records in manageable chunks within the execute method.

1. **Scenario: How would you implement error handling and logging in a Batch Apex job?**

**Answer:**

* + Try-Catch Blocks: Use try-catch blocks within the execute method to handle errors.
  + Stateful Interface: Implement the Database.Stateful interface to maintain state and log errors.
  + Finish Method: Log any final errors or perform cleanup operations in the finish method.

**Error in Salesforce**

**Common Salesforce Errors**

1. **System.LimitException**: Too many SOQL queries: 101Explanation: This error occurs when a single transaction exceeds the limit of 100 SOQL queries.Solution: Optimize your code to reduce the number of SOQL queries, use collections, and avoid queries inside loops.
2. **System.LimitException:** Too many DML statements: 151Explanation: This error occurs when a single transaction exceeds the limit of 150 DML statements.Solution: Bulkify your code to perform DML operations on collections of records instead of individual records.
3. **System.QueryException:** List has no rows for assignment to SObjectExplanation: This error occurs when a SOQL query returns no records, but the result is assigned to a single SObject variable.Solution: Use a list to store the query results and check if the list is empty before accessing its elements.
4. **System.DmlException**: Insert failed. First exception on row 0; first error: DUPLICATE\_VALUEExplanation: This error occurs when trying to insert a record with a unique field value that already exists.Solution: Check for existing records with the same unique field value before inserting a new record.
5. **System.CalloutException**: You have uncommitted work pending. Please commit or rollback before calling outExplanation: This error occurs when a callout is made after DML operations in the same transaction.Solution: Perform callouts before any DML operations or use future methods to handle callouts asynchronously.
6. **System.AsyncException**: Future method cannot be called from a future or batch methodExplanation: This error occurs when a future method is called from another future or batch method.Solution: Avoid chaining future methods. Use Queueable Apex for more complex asynchronous processing.
7. **System.StringException:** String length exceeds maximum: 131072Explanation: This error occurs when a string exceeds the maximum allowed length of 131,072 characters.Solution: Ensure that strings are within the allowed length or use Blob data type for larger text.
8. **System.NullPointerExceptionExplanation:** This error occurs when attempting to dereference a null object.Solution: Check for null values before accessing object properties or methods.
9. **System.TypeException:** Invalid conversion from runtime typeExplanation: This error occurs when trying to cast an object to an incompatible type.Solution: Ensure that the object being cast is of the expected type.
10. **System.DmlException:** Update failed. First exception on row 0; first error: FIELD\_CUSTOM\_VALIDATION\_EXCEPTIONExplanation: This error occurs when a custom validation rule fails during a DML operation.Solution: Review and adjust the validation rules or ensure that the data being updated meets the validation criteria.

**Advanced Errors**

1. **System.LimitException:** Apex CPU time limit exceededExplanation: This error occurs when the CPU time limit for a transaction is exceeded.Solution: Optimize your code to reduce CPU usage, avoid complex logic in loops, and use efficient algorithms.
2. **System.LimitException:** Too many future calls: 51Explanation: This error occurs when a single transaction exceeds the limit of 50 future method calls.Solution: Limit the number of future calls and consider using Queueable Apex for better control.
3. **System.DmlException:** Insert failed. First exception on row 0; first error: FIELD\_INTEGRITY\_EXCEPTIONExplanation: This error occurs when a required field is missing or a field value does not meet integrity constraints.Solution: Ensure all required fields are populated and field values meet integrity constraints before performing DML operations.
4. **System.QueryException:** Non-selective query against large object typeExplanation: This error occurs when a query is not selective enough and scans too many records.Solution: Use selective filters, indexed fields, and avoid full table scans in your queries.
5. **System.DmlException:** Insert failed. First exception on row 0; first error: CANNOT\_INSERT\_UPDATE\_ACTIVATE\_ENTITYExplanation: This error occurs when a trigger or process fails during a DML operation.Solution: Review and debug the trigger or process to identify and fix the issue.
6. **System.LimitException:** Too many callouts: 101Explanation: This error occurs when a single transaction exceeds the limit of 100 callouts.Solution: Reduce the number of callouts or batch them if possible.
7. **System.DmlException:** Insert failed. First exception on row 0; first error: INVALID\_CROSS\_REFERENCE\_KEYExplanation: This error occurs when a foreign key reference is invalid.Solution: Ensure that all foreign key references are valid and the related records exist.
8. **System.DmlException**: Insert failed. First exception on row 0; first error: REQUIRED\_FIELD\_MISSINGExplanation: This error occurs when a required field is missing during a DML operation.Solution: Ensure all required fields are populated before performing DML operations.
9. **System.DmlException:** Insert failed. First exception on row 0; first error: INVALID\_FIELD\_FOR\_INSERT\_UPDATEExplanation: This error occurs when trying to insert or update a field that is not allowed.Solution: Ensure that only allowed fields are included in the DML operation.
10. **System.DmlException:** Insert failed. First exception on row 0; first error: FIELD\_FILTER\_VALIDATION\_EXCEPTIONExplanation: This error occurs when a field value does not meet the criteria defined by a filter.Solution: Ensure that field values meet the criteria defined by any filters before performing DML operations.

**21.The Mixed DML Operation error:** in Salesforce occurs when you try to perform DML operations on certain combinations of SObjects in the same transaction. Specifically, it happens when you mix DML operations on setup objects (like User, Profile, Group, etc.) with non-setup objects (like Account, Contact, Opportunity, etc.) in a single transaction.

**Best Practice**

**OmniStudio DataRaptors read or write Salesforce SObject data or perform single-step data  
transformations. OmniStudio Integration Procedures can interact with many types of data and process it in  
multiple steps.**

**Use a single DataRaptor if:  
•** You need to read data from SObjects or write data to SObjects, but not both.  
• The SObjects you need to read from or write to have a defined relationship. For example, Accounts and  
Contacts have a relationship because a Contact can have an AccountId value.  
• You only need to work with JSON or XML data. No SObjects are involved.  
• You can perform any needed filtering, calculation, or reformatting of data values using one or a series of  
formulas.  
• You can make any needed changes to the data structure by mapping input JSON nodes to output JSON  
nodes.  
• Use unique names for OmniScript elements and Data Mapper response nodes.  
• Create targeted Data Mappers that only extract or load the data needed for one operation.  
• Use relationship notation (queries) whenever possible to pull data from other SObjects. For more information, see Relationship Notation versus Multiple Extract Steps.  
• To avoid performance issues on the server, keep the number of SObjects to three or fewer.  
• Ensure that all filtering and sorting (ORDER BY) operations are on indexed fields. The Id and Name fields are always indexed. For more information, see Indexes in Salesforce Help.  
• Use caching to store frequently accessed, infrequently updated data. See Cache for Omnistudio Data Mappers and Integration Procedures.

**Use an Integration Procedure if:**

**•** You need to both read from and write to one or more SObjects, which means you need to call at least two  
DataRaptors.  
• The SObjects you need to read from and/or write to have no defined relationship.  
• Transforming your data can't be done using formulas alone. For example, different conditions determine  
whether some filtering or calculations are performed at all.  
• JSON node mappings aren't straightforward and/or require a series of steps.  
• You need to read from or write to multiple data source types, such as SObjects, CSV files, external  
objects, Apex classes, and so on.  
• You need to perform actions such as calling a REST API, sending an email, merging lists, handling  
errors, and so on  
• Start with a clear understanding of the business strategy and roadmap.  
• Decide on the integration pattern and approach.  
• Collaborate with relevant stakeholders.  
• Whiteboard/brainstorm all possible business scenarios.  
• Choose the right implementation team.  
• Determine the integration type (Pull/Push).  
• Select the integration platform/tool that supports both current and future business requirements.  
• Make use of relevant Salesforce features in conjunction with the integration tools for better performance.  
• Make an informed decision on data storage/security/sync frequency.  
• Assess the scalability and maintainability of the integration solution.  
• Create a data and process integrity checklist.  
• Review and refine the project plan, timeline and cost.

**Best Pratcice of IP**• Use Integration Procedures for all data calls to Salesforce.  
• Use Try catch block in integration procedure instead of writing apex code  
• Use a Response Action to trim the data and only return what is needed.  
• Use multiple Response Actions with different Execution Conditional Formulas to allow an Integration Procedure to exit early under appropriate conditions.  
• Use caching to store frequently accessed, infrequently updated data.  
• To run data operations asynchronously, call Integration Procedures using these settings:  
Use Future — Use when the calling OmniScript or Integration Procedure doesn’t need a response and completion time is not critical.  
Invoke Mode: Fire and Forget — Use instead of Use Future when the calling OmniScript must invoke the Integration Procedure immediately.  
Invoke Mode: Non-Blocking — Use to run the Integration Procedure immediately while continuing the user interaction of the calling OmniScript. A response is returned when the Integration Procedure is complete

**OmniScript Best Practices**

• Use one owner for each OmniScript.  
• Identify reusable elements by building a skeleton of the entire OmniScript.  
• Document the purpose of an element in the element's Internal Notes property.  
• Maintain DataRaptors and Apex classes by avoiding element name changes. If the element name must  
be updated, apply the name changes to the DataRaptor or Apex class.  
• Avoid assigning a ContextId within the OmniScript. OmniScript's ContextId is a reserved key that assigns  
a Record Id from the URL.  
• When processes are repeatable across multiple OmniScripts, create a reusable OmniScript, and add it to the appropriate parent OmniScripts. See Embed an OmniScript in Another OmniScript.

**flexCard best Practices**

1.OmniStudio auto-compiles a LWC every time you activate your FlexCard, therefor there are some limitations in regards to the size of that auto-generated component. Each file within an LWC component folder cannot exceed 131 072 characters. FlexCards allow for nesting child FlexCards and those are separate LWCs, so you should/ must consider creating multiple smaller FlexCards to reduce the chances of hitting this limit.  
2.Custom LWCs increase the size of the auto-generated HTML a lot due to no dynamic import support on platform.  
3.Make your event listeners unique to avoid firing the same event handler more than once! Use the option to concatenate the {recordId} and other context variables.  
4.Auto-generated LWCs start with c-cf-\*\*  
5.Do not modify the auto-generated LWCs

**FLOW Best Practices**

1. Always Test Your Flows  
2. Consider Using Subflows  
3. Never Perform DML Statements in Loops  
4. Document Your Flows  
5. Never Hard Code Ids (Use Constants if You Must)  
6. Plan for Faults (and Handle Them)  
7. Utilize Before-Save Flows for Same Record Updates  
8. Make Use of Schedule-Triggered Flow and Asynchronous Paths  
9. Flow Isn’t Always the Best Idea  
10. If You Get Stuck, Ask the Trailblazer Community!

**Trigger Best Practices**

1. Bulkify Your Code  
2. Avoid DML/SOQL Queries in Loops  
3. Avoid Hard-Coded IDs  
4. Explicitly Declare Sharing Model  
5. Use a Single Trigger per SObject Type  
6. Modularize Your Code  
7. Test Multiple Scenarios  
8. Avoid Nested Loops  
9. Have a Naming Convention

**Best Practice of Apex**

1. Bulkify Apex Code  
2. Avoid SOQL & DML inside for Loop  
3. Querying Large Data Sets  
4. Use of Map of Sobject  
5. Use of the Limits Apex Methods  
6. Avoid Hardcoding IDs  
7. Use Database Methods while doing DML operation  
8. Exception Handling in Apex Code  
9. Write One Trigger per Object per event  
10. Use Asynchronous Apex  
11. Security and Sharing in Apex Code  
12. Make reusability of Apex Code  
13. Code coverage  
14. Return Early Pattern  
15. Avoid nesting loops within loops  
16. Don’t mix Apex, Process Builders, Workflow Rules, and Record-Triggered flows  
17. Naming Conventions.  
18. Setup Code review checklist and Code Review process

**Child to Parent Data Access.**

List<Contact> contacts = [SELECT Name, Account.Name, Account.Phone FROM Contact Where Account.Name='Jhon'];  
for (Contact con : contacts) {  
    System.debug('Contact Name: ' + con);  
    System.debug('Account Name: ' + con.Account);  
}

****

**Parent to Child data Access**

List<Account> accounts = [SELECT Name, (SELECT Name FROM Contacts) FROM Account Where Name='Jhon'];  
for (Account acc : accounts) {  
    System.debug(acc);  
    System.debug(acc.Contacts);  
}

****

**When Account Phone is Updated then update all Contact Phone by Map**

trigger UpdateContactPhone on Account (after update) {  
    // Map to hold account IDs and their new phone numbers  
**Map<Id, String> accountPhoneMap = new Map<Id, String>();**  
  
    // Populate the map with account IDs and their new phone numbers  
    for (Account acc : Trigger.new) {  
        Account oldAcc = Trigger.oldMap.get(acc.Id);  
        if (acc.Phone != oldAcc.Phone) {  
            **accountPhoneMap.put(acc.Id, acc.Phone);**        }  
    }  
  
    // If there are any accounts with updated phone numbers  
    if (!accountPhoneMap.isEmpty()) {  
        // Query contacts related to the accounts with updated phone numbers  
        **List<Contact> contactsToUpdate = [SELECT Id, Phone, AccountId FROM Contact WHERE AccountId IN :accountPhoneMap.keySet()];**  
        // Update the phone numbers of the contacts  
        for (Contact con : contactsToUpdate) {  
**con.Phone = accountPhoneMap.get(con.AccountId);**        }  
  
        // Perform the update  
        if (!contactsToUpdate.isEmpty()) {  
            update contactsToUpdate;  
        }  
    }  
}

**When Account Phone is Updated then update all Contact Phone by Map**

trigger UpdatePhoneOnContact on Account (after Update){  
    if(Trigger.IsAfter && Trigger.IsUpdate){  
 **Map<Id,String> AllAccountId = new Map<Id,String>();**  
        for(Account accNew : Trigger.New){  
            Account accOld = Trigger.oldMap.get(accNew.Id);  
            if(accold.Phone != accNew.Phone){  
                **AllAccountId.Put(accNew.Id , accNew.Phone);**  
  
            }  
        }  
        **List<Account> ContactWithAccount = [SELECT Id, Phone, (SELECT Id, Phone FROM Contacts) FROM Account WHERE Id IN: AllAccountId.keySet()];**  
        List<Contact> ContactPhoneUpdateList = new List<Contact>();  
        for(Account acc: ContactWithAccount){  
            for(Contact con: acc.Contacts){  
                **con.Phone =AllAccountId.get(acc.Id);**  
                ContactPhoneUpdateList.add(con);  
            }  
        }  
  
        If(ContactPhoneUpdateList.size() > 0){  
            UPDATE ContactPhoneUpdateList;  
        }  
    }  
}

**Here are some common scenarios where you might hit the heap size limit in Salesforce:**

1. **Handling Large Datasets**: If your Apex code processes a large number of records at once, such as querying thousands of records and storing them in a list, you can quickly exceed the heap size limit. For example, querying all contacts in a large organisation and storing them in a list without pagination.
2. **Storing Large Collections:** Using large collections like lists, maps, or sets to store data can consume a significant amount of memory. For instance, if you store a large number of records in a map for quick access, it can lead to heap size issues.
3. **Recursive Methods:** Recursive methods that accumulate data without proper clearing or batching can also cause heap size problems. For example, a recursive method that processes child records and keeps adding them to a list without clearing the list after each recursion.
4. **Complex Object Structures:** When dealing with complex object structures, such as objects with many related child objects, the memory usage can increase significantly. For example, retrieving accounts with all their related contacts and opportunities in a single query.
5. **Large JSON or XML Parsing:** Parsing large JSON or XML responses and storing the parsed data in memory can quickly exceed the heap size limit. For example, parsing a large JSON response from an external API and storing the data in a list of custom objects.
6. **Visualforce Page State:** Storing too much data in the view state of a Visualforce page can also lead to heap size issues. For example, if you store a large list of records in a controller variable that is part of the view state.

**To avoid hitting the heap size limit, you can follow best practices such as:**

* **Optimize SOQL Queries**: Retrieve only the fields you need and use SOQL for loops to process large datasets in smaller chunks.
* **Use Pagination:** Display data in smaller chunks using pagination in Visualforce pages or Lightning components.
* **Asynchronous Processing:** Use Batch Apex or Queueable Apex to process large volumes of records in smaller, manageable tasks.
* **Clear Variables:** Set large collections to null once they are no longer needed to free up memory.
* **Efficient Data Structures:** Use efficient data structures and avoid storing excessive data in collections.

**In Salesforce, triggers can be defined to run before or after certain events on a record, such as insert, update, delete, etc. When you define a trigger to run after an event (like after update), the records being processed are already saved to the database. This means they are in a read-only state and cannot be modified directly within the after trigger.**

**Here's a breakdown:**

* **Before triggers:** These run before the record is saved to the database. You can modify the record's fields in these triggers because the record hasn't been committed to the database yet.
* **After triggers:** These run after the record has been saved to the database. At this point, the record is read-only, and any attempt to modify it directly will result in the System.FinalException: Record is read-only error.

**For example, if you have a trigger like this:**

trigger AccountAfterUpdate on Account (after update) {  
    for (Account acc : Trigger.new) {  
        acc.Name = 'New Name'; // This will cause the read-only error  
    }  
}

**To avoid this error, you should use a before update trigger if you need to modify the record:**

trigger AccountBeforeUpdate on Account (before update) {  
    for (Account acc : Trigger.new) {  
        acc.Name = 'New Name'; // This is allowed  
    }  
}

**If you need to perform actions after the record is saved, you can use after triggers to perform operations that don't involve modifying the record directly, such as logging or making callouts.**

| **Trigger Event** | **Trigger.new** | **Trigger.old** |
| --- | --- | --- |
| **before insert** | Yes | No |
| **after insert** | Yes | No |
| **before update** | Yes | Yes |
| **after update** | Yes | Yes |
| **before delete** | No | Yes |
| **after delete** | No | Yes |
| **after undelete** | Yes | No |

**To skip a validation rule for a specific profile in Salesforce**

1. **Identify the Profile:** First, identify the profile name or ID you want to exclude. You can find this by navigating to Setup → Profiles and noting down the profile name or its ID (found in the URL after /Profile/).
2. **Modify the Validation Rule Formula:** Update your validation rule to include a condition that excludes the selected profile. Here’s an example formula:

**NOT($Profile.Name = "System Administrator") && <Your Original Validation Rule>**

1. **Alternatively, you can use the Profile ID:**

**NOT($Profile.Id = "00eXXXXXXXXXXXX") && <Your Original Validation Rule>**

1. This formula ensures that the validation rule applies to all profiles except the specified one
2. Test Your Changes: After saving your changes, log in as a user with the excluded profile and perform actions that would typically trigger the validation rule. Verify that users with other profiles are still restricted by the rule

**Example Use Case:** If you have a validation rule that prevents saving Opportunities with a past Close Date, but you want System Administrators to bypass this rule, your formula would look like this:

**NOT($Profile.Name = "System Administrator") && CloseDate < TODAY()**

**This allows only System Administrators to save Opportunities with past Close Dates**

**Best Practices:**

* Use Profile IDs Over Names: Profile names may change, but IDs remain consistent.
* Document Your Changes: Add comments in the validation rule to explain why specific profiles are excluded.
* Test Thoroughly: Ensure the validation rule works as expected for all users.
* Minimize Exceptions: Use this technique sparingly to avoid compromising data quality has context menu

**@wire decorator**

Sure! **The @wire decorator** in Lightning Web Components (LWC) is used to read Salesforce data and call Apex methods. Here's a comprehensive overview:

**1. Basic Usage**

* **Importing Apex Methods**: You can import an Apex method using the @salesforce/apex module.

import apexMethod from '@salesforce/apex/Namespace.Classname.apexMethod';

* **Wiring to a Property**: Use @wire to bind the Apex method to a property.

import { LightningElement, wire } from 'lwc';  
import getContacts from '@salesforce/apex/ContactController.getContacts';  
  
export default class MyComponent extends LightningElement {  
    @wire(getContacts) contacts;  
}

**2. Wiring to a Function**

* **Handling Data and Errors**: You can wire the Apex method to a function to handle data and errors.

import { LightningElement, wire } from 'lwc';  
import getContacts from '@salesforce/apex/ContactController.getContacts';  
  
export default class MyComponent extends LightningElement {  
    @wire(getContacts)  
    wiredContacts({ error, data }) {  
        if (data) {  
            // Handle data  
        } else if (error) {  
            // Handle error  
        }  
    }  
}

**3. Passing Parameters**

* **Dynamic Parameters**: Pass parameters to the Apex method by using an object.

import { LightningElement, wire, track } from 'lwc';  
import getContacts from '@salesforce/apex/ContactController.getContacts';  
  
export default class MyComponent extends LightningElement {  
    @track searchKey = '';  
  
    @wire(getContacts, { searchKey: '$searchKey' }) contacts;  
}

**4. Refreshing Data**

* **Refreshing Wired Data**: Use refreshApex to refresh the data provisioned by the wire service.

import { LightningElement, wire } from 'lwc';  
import { refreshApex } from '@salesforce/apex';  
import getContacts from '@salesforce/apex/ContactController.getContacts';  
  
export default class MyComponent extends LightningElement {  
    contacts;  
  
    @wire(getContacts)  
    wiredContacts(result) {  
        this.contacts = result;  
    }  
  
    handleRefresh() {  
        refreshApex(this.contacts);  
    }  
}

**5. Best Practices**

* **Avoid Overloading**: Do not overload @AuraEnabled Apex methods as it can cause errors.
* **Immutable Data**: The data provided by the wire service is immutable. To mutate data, make a shallow copy.
* **Reactive Variables**: Use reactive variables (prefixed with $) to dynamically update parameters

**fetch data from Salesforce By LWC**

In Lightning Web Components (LWC), there are several ways to fetch data from Salesforce. Here are the main methods:

**1. Using @wire Decorator**

* **Apex Methods**: Fetch data by calling Apex methods.

import { LightningElement, wire } from 'lwc';  
import getContacts from '@salesforce/apex/ContactController.getContacts';  
  
export default class MyComponent extends LightningElement {  
    @wire(getContacts) contacts;  
}

* **Standard Salesforce Data**: Fetch standard Salesforce data using wire adapters like getRecord, getObjectInfo, etc.

import { LightningElement, wire } from 'lwc';  
import { getRecord } from 'lightning/uiRecordApi';  
  
export default class MyComponent extends LightningElement {  
    @wire(getRecord, { recordId: '001xx000003DGbYAAW', fields: ['Account.Name'] }) account;  
}

**2. Using Imperative Apex Calls**

* **Apex Methods**: Fetch data by calling Apex methods imperatively.

import { LightningElement } from 'lwc';  
import getContacts from '@salesforce/apex/ContactController.getContacts';  
  
export default class MyComponent extends LightningElement {  
    contacts;  
  
    connectedCallback() {  
        this.loadContacts();  
    }  
  
    loadContacts() {  
        getContacts()  
            .then(result => {  
                this.contacts = result;  
            })  
            .catch(error => {  
                console.error(error);  
            });  
    }  
}

**3. Using Lightning Data Service (LDS)**

* **Record Data**: Fetch record data using lightning/uiRecordApi.

import { LightningElement, wire } from 'lwc';  
import { getRecord } from 'lightning/uiRecordApi';  
  
export default class MyComponent extends LightningElement {  
    @wire(getRecord, { recordId: '001xx000003DGbYAAW', fields: ['Account.Name'] }) account;  
}

* **Object Info**: Fetch object metadata using getObjectInfo.

import { LightningElement, wire } from 'lwc';  
import { getObjectInfo } from 'lightning/uiObjectInfoApi';  
import ACCOUNT\_OBJECT from '@salesforce/schema/Account';  
  
export default class MyComponent extends LightningElement {  
    @wire(getObjectInfo, { objectApiName: ACCOUNT\_OBJECT }) accountMetadata;  
}

**4. Using SOQL Queries in Apex**

* **Custom SOQL Queries**: Write custom SOQL queries in Apex to fetch data.

public with sharing class ContactController {  
    @AuraEnabled(cacheable=true)  
    public static List<Contact> getContacts() {  
        return [SELECT Id, Name, Email FROM Contact];  
    }  
}

**5. Using REST API**

* **REST API Calls**: Make REST API calls from LWC to fetch data.

import { LightningElement } from 'lwc';  
  
export default class MyComponent extends LightningElement {  
    connectedCallback() {  
        fetch('/services/data/v50.0/query?q=SELECT+Id,Name+FROM+Account')  
            .then(response => response.json())  
            .then(data => {  
                console.log(data);  
            })  
            .catch(error => {  
                console.error(error);  
            });  
    }  
}

**6. Using GraphQL API**

* **GraphQL API Calls**: Make GraphQL API calls from LWC to fetch data.

import { LightningElement } from 'lwc';  
  
export default class MyComponent extends LightningElement {  
    connectedCallback() {  
        fetch('/services/data/v50.0/graphql', {  
            method: 'POST',  
            headers: {  
                'Content-Type': 'application/json'  
            },  
            body: JSON.stringify({ query: '{ account { id name } }' })  
        })  
            .then(response => response.json())  
            .then(data => {  
                console.log(data);  
            })  
            .catch(error => {  
                console.error(error);  
            });  
    }  
}

SELECT Id, TestValue\_\_c FROM Account WHERE TestValue\_\_c > 0 ORDER BY TestValue\_\_c desc LIMIT 1 OFFSET 1

Differences between calling an Apex method **imperatively** and using the @wire decorator (which is often referred to as the "normal" way).

**Imperative Call**

Imperative calls are made explicitly by the developer, usually in response to a user action like a button click. This approach gives you more control over when the Apex method is called.

**Example Code**

**Apex Class:**

public with sharing class MyApexClass {  
    @AuraEnabled  
    public static void updateAccount(Id accountId, String newName) {  
        Account acc = [SELECT Id, Name FROM Account WHERE Id = :accountId];  
        acc.Name = newName;  
        update acc;  
    }  
}

**LWC JavaScript:**

import { LightningElement } from 'lwc';  
import updateAccount from '@salesforce/apex/MyApexClass.updateAccount';  
  
export default class MyComponent extends LightningElement {  
    handleUpdate() {  
        updateAccount({ accountId: '001...', newName: 'New Account Name' })  
            .then(result => {  
                // Handle success  
                console.log('Account updated successfully');  
            })  
            .catch(error => {  
                // Handle error  
                console.error('Error updating account:', error);  
            });  
    }  
}

**LWC HTML:**

<template>  
    <lightning-button label="Update Account" onclick={handleUpdate}></lightning-button>  
</template>

**Wire Service (Normal Way)**

The @wire decorator is used to call Apex methods reactively. The method is called automatically when the component is initialized or when any of its reactive parameters change.

**Example Code**

**Apex Class:**

public with sharing class MyApexClass {  
    @AuraEnabled(cacheable=true)  
    public static List<Account> getAccounts() {  
        return [SELECT Id, Name FROM Account];  
    }  
}

**LWC JavaScript:**

import { LightningElement, wire } from 'lwc';  
import getAccounts from '@salesforce/apex/MyApexClass.getAccounts';  
  
export default class MyComponent extends LightningElement {  
    @wire(getAccounts)  
    accounts;  
}

**LWC HTML:**

<template>  
    <template if:true={accounts.data}>  
        <ul>  
            <template for:each={accounts.data} for:item="account">  
                <li key={account.Id}>{account.Name}</li>  
            </template>  
        </ul>  
    </template>  
    <template if:true={accounts.error}>  
        <p>Error loading accounts: {accounts.error}</p>  
    </template>  
</template>

**Explanation**

**Imperative Call:**

* **Control**: You have full control over when the method is called.
* **Use Case**: Ideal for actions triggered by user interactions, such as button clicks.
* **Flexibility**: Can handle both read and write operations (DML).

**Wire Service:**

* **Reactivity**: The method is called automatically based on component lifecycle and reactive parameters.
* **Use Case**: Best for fetching data that needs to be displayed when the component loads or when certain parameters change.
* **Read-Only**: Suitable for read-only operations, especially when using cacheable=true.

Both approaches have their own advantages and are suitable for different scenarios. Imperative calls offer more control and flexibility, while the wire service provides a more declarative and reactive way to fetch data.

**Asynchronous Apex in Salesforce**

**Asynchronous Apex in Salesforce** allows you to run processes in the background, which can be very useful for long-running operations. Here's a quick overview of the different types of Asynchronous Apex and when to use each:

1. **Future Methods**:
   * **When to Use**: For long-running operations that need to be executed asynchronously, such as callouts to external web services or segregating DML operations to avoid mixed DML errors.
   * **Example**:

@future  
public static void myFutureMethod() {  
    // Your code here  
}

1. **Queueable Apex**:
   * **When to Use**: For complex jobs that need to be chained or require monitoring. It allows passing complex types to the job.
   * **Example**:
   * **Chaining Jobs**:

public class FirstJob implements Queueable {  
    public void execute(QueueableContext context) {  
        // Your code here  
        System.enqueueJob(new SecondJob());  
    }  
}

public class MyQueueableClass implements Queueable {  
    public void execute(QueueableContext context) {  
        // Your code here  
    }  
}

1. **Batch Apex**:
   * **When to Use**: For operations that need to process large volumes of data in batches, such as database maintenance jobs.
   * **Example**:

public class MyBatchClass implements Database.Batchable<sObject> {  
    public Database.QueryLocator start(Database.BatchableContext context) {  
        return Database.getQueryLocator('SELECT Id FROM Account');  
    }  
    public void execute(Database.BatchableContext context, List<sObject> scope) {  
        // Your code here  
    }  
    public void finish(Database.BatchableContext context) {  
        // Your code here  
    }  
}

1. **Scheduled Apex**:
   * **When to Use**: For scheduling Apex classes to run at specific times.
   * **Example**:

public class MyScheduledClass implements Schedulable {  
    public void execute(SchedulableContext context) {  
        // Your code here  
    }  
}  
// Scheduling the job  
String cronExp = '0 0 12 \* \* ?'; // Every day at noon  
System.schedule('My Scheduled Job', cronExp, new MyScheduledClass());

AI-generated content may be incorre\\

1. **Future Methods**:
   * **Cannot be called from**: Another future method, batch job, or scheduled Apex.
   * **Can be called from**: Synchronous Apex (e.g., triggers, controllers).
2. **Queueable Apex**:
   * **Can be called from**: Synchronous Apex, future methods, batch jobs, and scheduled Apex.
   * **Chaining**: You can chain one Queueable job to another.
3. **Batch Apex**:
   * **Cannot be called from**: Future methods.
   * **Can be called from**: Synchronous Apex, Queueable Apex, and scheduled Apex.
   * **Chaining**: You can start another batch job from the finish method of a batch job.
4. **Scheduled Apex**:
   * **Can be called from**: Synchronous Apex, future methods, Queueable Apex, and batch jobs.
   * **Can schedule**: Batch jobs, Queueable jobs, and other scheduled jobs.

Here's a quick reference table:

| **Type** | **Can Call** | **Cannot Call** |
| --- | --- | --- |
| Future Methods | Synchronous Apex | Future Methods, Batch Jobs, Scheduled Apex |
| Queueable Apex | Synchronous Apex, Future Methods, Batch Jobs, Scheduled Apex | - |
| Batch Apex | Synchronous Apex, Queueable Apex, Scheduled Apex | Future Methods |
| Scheduled Apex | Synchronous Apex, Future Methods, Queueable Apex, Batch Jobs | - |

**Question**

**how to use one omniscript into another omniscript?**

**in setup reusable checkbox shoud be enabled and omniscript is Actie.**

**not able to see the flexcard in omniscript what was the region ?**

**in Flexcard setup Omniscript Support checkbox shoud be enabled and FlexCard is Actie.**

**Flex card**

**SOQL, SOSL, Dataraptor, Intergation Procedure, Custom, APEX remote, APEX Rest, Rest, Streaming API, SDK**

**Sometime we get this error**

**Error!**

**Unauthorized endpoint, please check Setup->Security->Remote site settings. endpoint = https://cognizant-3ac-dev-ed.develop.lightning.force.com/services/data/v46.0/tooling/composite/batch**

**Solution: jsut create a remote site setting because flexcard is working like LWC component and in this senerio it not able to find the endpoint of the page.**

**shift+w for close all the tab in omnistudio**

**linear gradient generator: for use of custom CSS**

**Datatable: use to show the data in table format**

**use LWC in Omniscript : omniscriptBaseMixin(for access all data use: this.omniJsonData)**

**for run IP from Class(Remote Action)**

**vlocity\_cmt.IntegrationProcedureService.runIntegrationService**

**for run DataMapper from Class**

**vlocity\_cmt.DRGlobal.processObjectsJSON**

**What is the difference between Extract and Turbo Extract?**

**In any case we have to use only one Object but we prefer Extract over Turbo extract why ?**

**We have two fields on Omniscript where we are getting Contact Name and DOB How to find Age of that Contact?**

**Difference between Edit Block and typeAhed Block ?**

**where we can Store Email and Phone details in OmniScript?**

**Can we use OmniScript Inside OmniScript ?**

**we have two fields and we have to make both as mandatory but when user enter any one of them then 2nd field will not going to mandatory?**

**how to use LWC component into OmniScript?**

**FlexCard DataSource?**

**can we use IP inside IP?**

**Inside Ip we have 5 Step and in 1st Step we get "Status = Completed" so we have to skip 2,3,4 Step and directly jump on 5th Step How we can do this ?**

**How to use Class inside IP ?**

**We have to show count of total Contact where Email Id = NULL (we are Passing AccountId so How we can build this data Mapper) ?**

**how to use one omniscript into another omniscript?**

**not able to see the flexcard in omniscript what was the region ?**

**How to use Flow inside IP ?**

**how to change standard object label in salesforce?**

**-->From Setup, in the Quick Find box, find and select Rename Tabs and Labels.**

**Trigger**

**Trigger**

**1. Upon Account Creation if industry is not null and having value as 'Media' then Populate Rating as 'Hot'**

**trigger AccountTrigger on Account (before insert) {**

**if(Trigger.isBefore && Trigger.isInsert){**

**for(Account acc: Trigger.New){**

**if(acc.Industry != 'null' && acc.Industry == 'Media'){**

**acc.Rating = 'Hot';**

**}**

**}**

**}**

**}**

**2.Upon Opportunity Creation if Amount is not null and is greater then 100000 then Populate 'Hot Opportunity' in the description Fields.**

**trigger OpportunityTrigger on Opportunity (before insert) {**

**if(Trigger.isBefore && Trigger.isInsert){**

**for(Opportunity Opp: Trigger.New){**

**if(Opp.Amount != null && Opp.Amount > 100000){**

**Opp.Description ='Hot Opportunity';**

**}**

**}**

**}**

**}**

**3. When an Account inserts and CopyBillingToShipping (Custom Field) CheckBox is Checked than automatically copy accout billing address into account shipping address.**

**trigger AccountTrigger on Account (before insert) {**

**if(Trigger.isBefore && Trigger.isInsert){**

**for(Account acc: Trigger.New){**

**if(acc.CopyBillingToShipping\_\_c == true){**

**acc.ShippingCity = acc.BillingCity;**

**acc.ShippingCountry= acc.BillingCountry;**

**acc.ShippingPostalCode = acc.BillingPostalCode;**

**acc.ShippingState = acc.BillingState;**

**acc.ShippingStreet= acc.BillingStreet;**

**}**

**}**

**}**

**}**

**3. When any contact description is Updated then related Account Description also got Updated.**

**trigger conDesc on Contact (after Update) {**

**if(Trigger.isAfter && Trigger.isUpdate){**

**String description;**

**Set<Id> ConId = new Set<Id>();**

**for(Contact connew: Trigger.new){**

**if(connew.Description != Trigger.oldMap.get(connew.id).Description ){**

**ConId.add(connew.AccountId);**

**description = connew.Description;**

**}**

**}**

**List<Account> conWithacc = [SELECT Id, Description FROM Account WHERE Id IN : ConId];**

**List<Account> accUpdate = new List<Account>();**

**for(Account acc: conWithacc){**

**acc.Description = description;**

**accUpdate.add(acc);**

**}**

**if(!accUpdate.isEmpty()){**

**UPDATE accUpdate;**

**}**

**}**

**}**

**4. When Account Phone is Updated then all Contacts Phone will Update (by MAP)**

**trigger UpdatePhoneOnContact on Account (after Update){**

**if(Trigger.IsAfter && Trigger.IsUpdate){**

**Map<Id,String> AllAccountId = new Map<Id,String>();**

**for(Account accNew : Trigger.New){**

**Account accOld = Trigger.oldMap.get(accNew.Id);**

**if(accold.Phone != accNew.Phone){**

**AllAccountId.Put(accNew.Id , accNew.Phone);**

**}**

**}**

**List<Account> ContactWithAccount = [SELECT Id, Phone, (SELECT Id, Phone FROM Contacts) FROM Account WHERE Id IN: AllAccountId.keySet()];**

**List<Contact> ContactPhoneUpdateList = new List<Contact>();**

**for(Account acc: ContactWithAccount){**

**for(Contact con: acc.Contacts){**

**con.Phone =AllAccountId.get(acc.Id);**

**ContactPhoneUpdateList.add(con);**

**}**

**}**

**If(ContactPhoneUpdateList.size() > 0){**

**UPDATE ContactPhoneUpdateList;**

**}**

**}**

**}**

**Using List**

**trigger UpdatePhoneOnContact On Account(after Update){**

**if(Trigger.isAfter && Trigger.isUpdate){**

**String Phone;**

**Set<Id> AccountId = new Set<Id>();**

**for(Account accNew: Trigger.New){**

**Account accOld = Trigger.oldMap.get(accNew.Id);**

**if(accNew.Phone != accOld.Phone){**

**Phone = accnew.Phone;**

**AccountId.add(accOld.Id);**

**}**

**}**

**List<Account> AccountwithContacts = [SELECT Id, Phone, (SELECT Id, Phone FROM Contacts) FROM Account WHERE Id IN: AccountId];**

**List<Contact> ContactstoUpdate = new List<Contact>();**

**for(Account acc: AccountwithContacts){**

**for(Contact con: acc.Contacts){**

**con.Phone =Phone;**

**ContactstoUpdate.add(con);**

**}**

**}**

**if(!ContactstoUpdate.isEmpty()){**

**UPDATE ContactstoUpdate;**

**}**

**}**

**}**

**without using SOQL quary**

**trigger Account on Account (after Update) {**

**if(Trigger.isAfter && Trigger.isUpdate){**

**Map<id, String> AccountIdwithPhone = new Map<Id, String>();**

**for(Account accNew: Trigger.new){**

**Account accOld = Trigger.oldMap.get(accNew.Id);**

**if(accNew.Phone != accOld.Phone)**

**AccountIdwithPhone.put(accOld.Id, accNew.Phone);**

**}**

**List<Contact> contactList = [SELECT Id, AccountId FROM Contact WHERE AccountId IN:AccountIdwithPhone.keySet()];**

**List<Contact> contactswithUpdatePhoneNumber = new List<Contact>();**

**for(Contact con: contactList){**

**con.Phone= AccountIdwithPhone.get(con.AccountId);**

**contactswithUpdatePhoneNumber.add(con);**

**}**

**if(contactswithUpdatePhoneNumber.Size()> 0){**

**UPDATE contactswithUpdatePhoneNumber;**

**}**

**}**

**}**

**5. When Contact Phone is Updated then Account Phone will Update (by MAP)**

**trigger updatePhoneOnAccount on Contact (after Update) {**

**if(trigger.isAfter && Trigger.isUpdate){**

**Map<Id, String> contactIdswithPhone = new Map<Id, String>();**

**for(Contact conNew: Trigger.new){**

**Contact conOld = Trigger.oldMap.get(conNew.Id);**

**if(conOld.Phone != conNew.Phone){**

**contactIdswithPhone.put(conOld.AccountId, conNew.Phone);**

**}**

**}**

**List<Account> accountrecords = [SELECT Id, Phone FROM Account WHERE Id IN : contactIdswithPhone.keySet()];**

**List<Account> updatedAccountwithPhone = new List<Account>();**

**for(Account acc: accountrecords){**

**acc.Phone = contactIdswithPhone.get(acc.Id);**

**// acc.Phone = acc.Phone; (it is also working because we are getting Phone value from SOQL Quary)**

**updatedAccountwithPhone.add(acc);**

**}**

**if(updatedAccountwithPhone.Size()>0){**

**UPDATE updatedAccountwithPhone;**

**}**

**}**

**}**

**by using List**

**trigger updatePhoneOnAccount on Contact (after Update) {**

**if(trigger.isAfter && Trigger.isUpdate){**

**Set<Id> ContactIds = new Set<Id>();**

**String Phone;**

**for(Contact conNew: Trigger.new){**

**Contact conOld = Trigger.oldMap.get(conNew.Id);**

**if(conOld.Phone != conNew.Phone){**

**Phone = conNew.Phone;**

**ContactIds.add(conOld.AccountId);**

**}**

**}**

**List<Account> accountrecords = [SELECT Id, Phone FROM Account WHERE Id In : ContactIds];**

**List<Account> updatedAccountwithPhone = new List<Account>();**

**for(Account acc: accountrecords){**

**acc.Phone = Phone;**

**updatedAccountwithPhone.add(acc);**

**}**

**if(updatedAccountwithPhone.Size()>0){**

**UPDATE updatedAccountwithPhone;**

**}**

**}**

**}**

**6. Update Account Description with Case Number when any New Case is Created**

**trigger Cases on Case (after insert) {**

**if(Trigger.isAfter && Trigger.isInsert){**

**for(Case cs: Trigger.new){**

**if(cs.AccountId!= null){**

**Account acc = new Account();**

**acc.Description= cs.CaseNumber;**

**acc.Id= cs.AccountId;**

**UPDATE acc;**

**}**

**}**

**}**

**}**

**7. if both checkbox are true then create Contact and Opportunity and create opportunity when account Active ="Yes"**

**trigger Account on Account (after insert) {**

**if(Trigger.isAfter && Trigger.isInsert){**

**for(Account acc: Trigger.new){**

**if(acc.AContact\_\_c== true){**

**Contact con= new Contact();**

**con.AccountId = acc.Id;**

**con.LastName = acc.Name +' '+'A';**

**Insert Con;**

**}**

**if(acc.AOpportunity\_\_c == true && acc.Active\_\_c == 'Yes'){**

**Opportunity opp = new Opportunity();**

**opp.Name = acc.Name;**

**opp.CloseDate = System.Today() + 5;**

**opp.StageName = 'Closed Won';**

**opp.AccountId = acc.Id;**

**Insert opp;**

**}**

**}**

**}**

**}**

**8. Upon Account Creation or updation if industry is not null and having value as 'Media' then Populate Rating as 'Hot'**

**trigger Account on Account (before insert, before Update) {**

**if((Trigger.isBefore && Trigger.isInsert) || (Trigger.isBefore && Trigger.isUpdate)){**

**for(Account acc: Trigger.new){**

**if(acc.Industry != Null && acc.Industry == 'Media'){**

**acc.Rating ='Hot';**

**}**

**}**

**}**

**}**

**9 Create Task with related Opportunity when stage is Changed.**

**trigger Opportunity on Opportunity (after Update) {**

**if(Trigger.isAfter && Trigger.isUpdate){**

**for(Opportunity oppNew: Trigger.new){**

**Opportunity oppOld = Trigger.oldMap.get(oppNew.Id);**

**if(oppNew.StageName != oppOld.StageName){**

**Task ts = new Task();**

**ts.Status ='In Progress';**

**ts.Subject ='Call';**

**ts.Priority = 'High';**

**ts.WhatId = oppNew.Id;**

**ts.OwnerId = UserInfo.getUserId();**

**Insert ts;**

**}**

**}**

**}**

**}**

**10. when Account Active filed changes from yes to No make all Opportunity files Closed Lost if stage not equal Close Won**

**trigger Account on Account (after Update) {**

**if(Trigger.isAfter && Trigger.isUpdate){**

**Set<Id> AccountIds = new Set<Id>();**

**for(Account accNew: Trigger.new){**

**Account accOld = Trigger.oldMap.get(accNew.Id);**

**if(accOld.Active\_\_c == 'Yes' && accNew.Active\_\_c == 'No')**

**AccountIds.add(accNew.Id);**

**System.debug(AccountIds);**

**}**

**List<Account> AccountwithOpportunity = [SELECT Id,Active\_\_c, (SELECT Id, StageName FROM Opportunities) FROM Account WHERE Id IN: AccountIds];**

**System.debug(AccountwithOpportunity);**

**List<Opportunity> oppList = new List<Opportunity>();**

**for(Account acc: AccountwithOpportunity){**

**for(Opportunity opp: acc.Opportunities){**

**if(opp.StageName != 'Closed Won')**

**opp.StageName = 'Closed Lost';**

**oppList.add(opp);**

**}**

**}**

**if(!oppList.isEmpty())**

**UPDATE oppList;**

**}**

**}**

**11. Duplicate Account not create and send error message**

**trigger Account on Account (before Insert) {**

**if(Trigger.isBefore && Trigger.isInsert){**

**Set<String> accountName = new Set<String>();**

**for(Account acc: Trigger.new){**

**accountName.add(acc.Name);**

**}**

**List<Account> accountDetails = [SELECT Name FROM Account WHERE Name IN: accountName];**

**for(Account acc: Trigger.new){**

**if(accountDetails.Size()> 0){**

**acc.addError('Duplicate Account');**

**}**

**}**

**}**

**}**

**12. Active account can not be Delete**

**trigger Account on Account (before delete) {**

**if(Trigger.isBefore && Trigger.isDelete){**

**for(Account acc: Trigger.old){**

**if(acc.Active\_\_c == 'Yes'){**

**acc.addError('Active Account can\'t delete');**

**}**

**}**

**}**

**}**

**13. Only System Admin can delete Account**

**trigger Account on Account (before delete) {**

**if(Trigger.isBefore && Trigger.isDelete){**

**Profile p = [SELECT Id FROM Profile WHERE Name= 'System Administrator'];**

**for(Account acc: Trigger.old){**

**if(UserInfo.getProfileId() != P.Id){ // UserInfo.getProfileId() for UserId who is Login.**

**acc.addError('Only System Administrator Can Delete this Account');**

**}**

**}**

**}**

**}**

**14. If any Account have Opportunity then we are not able to delete that account**

**trigger Account on Account (before delete) {**

**if(Trigger.isBefore && Trigger.isDelete){**

**List<Opportunity> OppList = [SELECT Id FROM Opportunity WHERE AccountId IN: Trigger.oldMap.keySet()];**

**if(OppList.Size()> 0){**

**for(Account acc: Trigger.old){**

**acc.addError('Can not delete this Account because this Account have Opportunity');**

**}**

**}**

**}**

**}**

**15. If any Account have Case then we are not able to delete that account**

**trigger Account on Account (before delete) {**

**if(Trigger.isBefore && Trigger.isDelete){**

**List<Case> caseIds = [SELECT Id FROM Case WHERE AccountId IN: Trigger.oldMap.keySet()];**

**if(caseIds.Size()>0){**

**for(Account acc: Trigger.old){**

**acc.addError('Account have Case so it\'s can not be deleted');**

**}**

**}**

**}**

**}**

**16. Count all the Case Present on Account**

**trigger Account on Account (before update) {**

**if(Trigger.isBefore && Trigger.isUpdate){**

**List<Case> caseIds = [SELECT Id FROM Case WHERE AccountId IN: Trigger.newMap.keySet()];**

**for(Account acc: Trigger.new){**

**acc.Description = String.ValueOf(CaseIds.Size());**

**}**

**}**

**}**

**trigger Opportunity on Opportunity (after insert, after delete, after update, after Undelete) {**

**if(Trigger.isAfter && (Trigger.isInsert || Trigger.isUpdate || Trigger.isUndelete)){**

**INTEGER CountOpp =0;**

**Set<Id> AccountIds = new Set<Id>();**

**for(Opportunity opp: Trigger.new){**

**if(opp.AccountId != Null)**

**AccountIds.add(opp.AccountId);**

**}**

**if(Trigger.isAfter && Trigger.isDelete){**

**for(Opportunity opp: Trigger.old){**

**if(opp.AccountId != Null)**

**AccountIds.add(opp.AccountId);**

**}**

**List<Account> accWithopp = [SELECT Id, (SELECT Id FROM Opportunities) FROM Account WHERE Id IN: AccountIds];**

**List<Account> accFinal = new List<Account>();**

**for(Account acc: accWithopp){**

**for(Opportunity opp: acc.Opportunities){**

**CountOpp++;**

**}**

**if(CountOpp >=0){**

**acc.Description = String.Valueof(CountOpp);**

**accFinal.add(acc);**

**}**

**}**

**if(accFinal.Size()> 0)**

**UPDATE accFinal;**

**}**

**}**

**if(Trigger.isAfter && Trigger.isDelete){**

**INTEGER CountOpp =0;**

**Set<Id> AccountIds = new Set<Id>();**

**for(Opportunity opp: Trigger.old){**

**if(opp.AccountId != Null)**

**AccountIds.add(opp.AccountId);**

**}**

**List<Account> accWithopp = [SELECT Id, (SELECT Id FROM Opportunities) FROM Account WHERE Id IN: AccountIds];**

**List<Account> accFinal = new List<Account>();**

**for(Account acc: accWithopp){**

**for(Opportunity opp: acc.Opportunities){**

**CountOpp++;**

**}**

**if(CountOpp >=0){**

**acc.Description = String.Valueof(CountOpp);**

**accFinal.add(acc);**

**}**

**}**

**if(accFinal.Size()> 0)**

**UPDATE accFinal;**

**}**

**}**

**17. Write a trigger when Account Undelete then Active ='True'**

**trigger Account on Account (after undelete) {**

**if(Trigger.isAfter && Trigger.isUnDelete){**

**for(Account acc: Trigger.new){**

**acc.Active\_\_c = 'Yes';**

**}**

**UPDATE Trigger.new;**

**}**

**}**

**Use handller class for count of Contacts**

**//Trigger**

**trigger Contact on Contact (after insert) {**

**if(Trigger.isInsert && Trigger.isAfter){**

**ContactTriggerHandler.updateCount(Trigger.new);**

**}**

**}**

**//Handler Class**

**public class ContactTriggerHandler {**

**public static void updateCount(List<Contact> ConList){**

**Set<Id> accountId = new Set<Id>();**

**for(Contact con: ConList){**

**accountId.add(con.AccountId);**

**}**

**List<Account> AccountDetails = [SELECT id,Count\_\_c FROM Account WHERE Id IN: accountId];**

**for(Account acc: AccountDetails){**

**acc.count\_\_c = acc.Count\_\_c+ 1;**

**Update acc;**

**}**

**}**

**}**

**Omnistudio**

**import { LightningElement } from 'lwc';**

**import { OmniscriptBaseMixin } from 'omnistudio/omniscriptBaseMixin';**

**import { NavigationMixin } from 'lightning/navigation';**

**import pubsub from "omnistudio/pubsub";**

**export default class TalcottAgentSearchList extends OmniscriptBaseMixin(NavigationMixin(LightningElement)) {**

**pageReference;**

**data = [];**

**columns = [**

**{ label: 'Name', fieldName: 'Name' },**

**{ label: 'Phone', fieldName: 'Phone' },**

**{ label: 'Address', fieldName: 'Address' },**

**{**

**type: "button", label: 'Detail', initialWidth: 100, typeAttributes: {**

**label: 'Detail',**

**name: 'Detail',**

**title: 'Detail',**

**disabled: false,**

**value: 'Detail',**

**variant:'Neutral'**

**}**

**}**

**];**

**connectedCallback() {**

**this.data = this.omniJsonData?.AgentSearchTable;**

**pubsub.register('omniscript\_action', {**

**data: this.handleOmniActionData.bind(this),**

**});**

**//code**

**}**

**handleOmniActionData(data) {**

**this.data = this.omniJsonData?.AgentSearchTable;**

**console.log('Data => ' + JSON.stringify(data));**

**}**

**navigate(event){**

**this.pageReference = {**

**type: 'standard\_\_component',**

**attributes: {**

**componentName: 'omnistudio\_\_vlocityLWCOmniWrapper'**

**},**

**state: {**

**c\_\_target: 'c:talcottAgentAdvanceSearchViewDataEnglish',**

**c\_\_layout: 'lightning', // or can be 'newport'**

**c\_\_tabIcon: 'custom:custom18',**

**c\_\_tabLabel: 'Talcott\_AgentAdvanceSearchViewData',**

**c\_\_ContextId: event.detail.row.Id**

**}**

**};**

**console.log('entry')**

**this[NavigationMixin.GenerateUrl](this.pageReference).then(url => {**

**window.open(url, "\_blank");**

**});**

**}**

**}**

**Class**

**public class DataRaptorHelper implements Callable {**

**public Object call(String action, Map<String,Object>args){**

**Map<String,Object>input=(Map<String,Object>)args.get('input');**

**Map<String,Object>output=(Map<String,Object>)args.get('output');**

**Map<String,Object>options=(Map<String,Object>)args.get('options');**

**return invokeMethod(action,input,output,options);**

**}**

**public boolean invokeMethod(String methodName, Map<String,Object> input,Map<String,Object> output,Map<String,Object> options)**

**{**

**try{**

**if(methodName.equalsIgnoreCase('getTotalAccountsCount'))**

**{**

**List<Object> arg = (List<Object>)input.get('arguments');**

**output.put('result',arg.size());**

**}**

**}**

**catch(Exception e){**

**return false;**

**}**

**return true;**

**}**

**}**

**Apex Questions:**

**Explain the Salesforce Governor Limits and how you manage them in your code.**

**Describe the different types of Apex triggers and their use cases.**

**How do you handle bulk data processing in Apex?**

**What are the best practices for writing efficient SOQL queries?**

**Explain the concept of asynchronous Apex and its use cases.**

**How do you implement exception handling in Apex?**

**Describe the use of custom metadata types in Apex.**

**How do you optimize Apex code for performance?**

**Explain the difference between a managed and unmanaged package.**

**How do you use Apex to integrate with external systems?**

**What are the different types of collections in Apex and their use cases?**

**How do you handle large data volumes in Salesforce using Apex?**

**Explain the use of custom settings and custom labels in Apex.**

**How do you write test classes in Apex and ensure high code coverage?**

**Describe a complex Apex project you have worked on and the challenges you faced.**

**LWC Questions:**

**What is the lifecycle of a Lightning Web Component (LWC)?**

**How do you handle data binding in LWC?**

**Explain the use of Lightning Data Service (LDS) in LWC.**

**How do you communicate between LWC components?**

**Describe the process of creating a custom LWC.**

**How do you handle events in LWC?**

**Explain the use of decorators (@api, @track, @wire) in LWC.**

**How do you implement security in LWC?**

**Describe the use of Lightning Message Service (LMS) for inter-component communication.**

**How do you optimize LWC for performance?**

**Explain the difference between LWC and Aura components.**

**How do you handle form validation in LWC?**

**Describe a scenario where you used LWC to solve a complex problem.**

**How do you integrate LWC with Apex?**

**Explain the use of custom events in LWC.**

**Scenario-Based Apex Questions:**

**A client wants to automatically assign leads to sales reps based on zip codes. How would you implement this in Salesforce?**

**You need to update a field on all Account records whenever an Opportunity closes as won. How would you achieve this using Apex?**

**Imagine you need to ensure that Contact records cannot be deleted if they are associated with an active Opportunity. How would you achieve this?**

**A business requires a custom object to track Customer Feedback. Each feedback record should be linked to a Contact, and users should only be able to submit feedback for Contacts they own. How would you set this up?**

**How would you design a solution for a requirement where sales managers must approve discounts over 30% on Opportunities, with the process initiating automatically when the discount is entered?**

**A client requests a daily report of all Opportunities closed in the last 24 hours emailed to all sales managers. How do you automate this?**

**How would you ensure that a custom field on Account is automatically populated with a unique identifier upon record creation?**

**You need to integrate Salesforce with an external system to fetch real-time data. How would you approach this integration using Apex?**

**Describe how you would handle a scenario where a batch job needs to process millions of records without hitting governor limits.**

**A client wants to implement a custom sharing rule that grants access to records based on a custom field value. How would you achieve this using Apex?**

**Scenario-Based LWC Questions:**

**You need to create a custom LWC that displays a list of Accounts and allows users to filter the list based on various criteria. How would you implement this?**

**Describe how you would handle communication between two LWC components that are not in the same component hierarchy.**

**A client wants a custom LWC form that validates user input before saving the data. How would you implement this validation?**

**You need to create a Lightning Web Component that interacts with a third-party API to fetch and display data. How would you approach this?**

**Explain how you would optimize a complex LWC for performance, especially when dealing with large datasets.**

**A business requires a custom LWC that allows users to drag and drop files for upload. How would you implement this functionality?**

**Describe how you would use Lightning Message Service (LMS) to enable communication between LWCs and Aura components.**

**You need to create a custom LWC that dynamically renders different components based on user input. How would you achieve this?**

**A client wants a custom LWC that displays real-time updates from a Salesforce object. How would you implement this using Lightning Data Service (LDS)?**

**Explain how you would handle error handling and user notifications in a complex LWC application.**

**PassWord**

**salesforce.com**

**sourabh@cohortnov2021.com**

**Sp2024@salesforce**

**sourabh@cohortnov2021.com**

**Sp2024@salesforce**

**sourabh.patel3-93786079396@industryapps.edu**

**GoIndustries#1**

**sourabhpatel@omni.com**

**Sp2024@salesforce1**

**sourabhpatel@omni.com**

**Sp2024@salesforce1**

**sourabh@cohortnov2021.com**

**Sp2024@salesforce**

**sourabh@digital.com**

**Sp13122024@2**

**wxyz123@gmail.com**

**Xyz@1234**

**sourabh@cohortnov2021.com**

**Sp2024@salesforce**

**sourabh.patel3-93786079396@industryapps.edu**

**GoIndustries#1**

**sourabh@cohortnov2021.com**

**Sp2024@salesforce**

**sourabh@digital.com**

**Sp13122024@1**

**sourabhpatel@omni.com**

**Sp2024@salesforce3**

**sourabhpatel@cpq.com**

**Sp2024@salesforce**

**sourabhpatel@epc-cpq.com**

**Sp2024@salesforce**

**subhajit.bose2-hpf5@force.com**

**Salesforce@12345**

**sourabh@cohortnov2021.com**

**Sp2024@salesforce**

**webassessor.com**

**sourabhpatel.it@gmail.com**

**Sourabh1999@**